conference

proceedings

2018 USENIX Annual Technical Conference

Boston, MA, USA

July 11–13, 2018

Sponsored by

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix
ASSOCIATION

Proceedings of the 2018 USENIX Annual Technical Conference

Boston, MA, USA July 11–13, 2018

ISBN 978-1-939133-02-1

USENIX ATC '18 Sponsors

Gold Sponsors

facebook

NSF

Silver Sponsors

NetApp

ORACLE

vmware

Bronze Sponsors

aws

IBM

Industry Partners and Media Sponsors

ACM *Queue*

Distributed Management Task Force (DMTF)

FreeBSD Foundation

No Starch Press

She Geeks Out

ISBN 978-1-939133-02-1

Thanks to Our USENIX Supporters

USENIX Patrons

Facebook Google Microsoft
NetApp Private Internet Access

USENIX Benefactors

Amazon Bloomberg Oracle
Squarespace VMware

USENIX Partners

Booking.com Can Stock Photo Cisco Meraki
Dealslands Fotosearch

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the
2018 USENIX Annual Technical Conference

July 11–13, 2018
Boston, MA, USA

Conference Organizers

Program Co-Chairs

Haryadi Gunawi, *University of Chicago*

Benjamin Reed, *Facebook*

Program Committee

Rachit Agarwal, *Cornell University*

Irfan Ahmad, *CachePhysics*

Deniz Altinbuken, *Google*

Mahesh Balakrishnan, *Yale University and Facebook*

Aruna Balasubramanian, *Stony Brook University*

Theophilus Benson, *Brown University*

John Bent, *Cray Inc.*

Pramod Bhatotia, *University of Edinburgh*

Alex Biryukov, *University of Luxembourg*

Dhruba Borthakur, *Rockset Inc.*

Nathan Bronson, *Facebook*

Irina Calciu, *VMware Research*

Michael Carbin, *MIT*

Vijay Chidambaram, *University of Texas at Austin*

David Chu, *Google*

Paolo Costa, *Microsoft Research*

Peter Desnoyers, *Northeastern University*

Yufei Ding, *University of California, Santa Barbara*

Pierre-Alain Fouque, *University of Rennes 1*

Lakshmi Ganesh, *Facebook*

Ada Gavrilovska, *Georgia Institute of Technology*

Phillipa Gill, *University of Massachusetts, Amherst*

Xiaohui (Helen) Gu, *North Carolina State University*

Indranil Gupta, *University of Illinois at Urbana-Champaign*

Dean Hildebrand, *Google*

Michio Honda, *NEC Laboratories Europe*

Yu Hua, Huazhong *University of Science and Technology*

Jian Huang, *University of Illinois at Urbana-Champaign*

Ryan Huang, *Johns Hopkins University*

H. Howie Huang, *George Washington University*

David Irwin, *University of Massachusetts, Amherst*

Junchen Jiang, *University of Chicago and Microsoft Research*

Flavio Junqueira, *Confluent*

Asim Kadav, *NEC Labs*

Konstantinos Karanasos, *Microsoft*

Samira Khan, *University of Virginia*

Taesoo Kim, *Georgia Institute of Technology*

Achmad Kistijantoro, *Institut Teknologi Bandung*

Julia Lawall, *Inria/LIP6*

I-Ting Angelina Lee, *Washington University in St. Louis*

Tanakorn Leesatapornwongsa, *Samsung Research America*

Cheng Li, *VMware*

Jinyang Li, *New York University*

Xing Lin, *NetApp*

Xiaosong Ma, *Qatar Computing Research Institute, HKBU, Qatar*

Dahlia Malkhi, *VMware Research*

Carlos Maltzahn, *University of California, Santa Cruz*

Changwoo Min, *Virginia Tech*

Iqbal Mohomed, *Samsung Research America*

Cristina Nita-Rotaru, *Northeastern University*

Vijayan Prabhakaran, *Amazon*

Christopher J. Rossbach, *University of Texas at Austin and VMware Research*

Russell Sears, *Pure Storage*

Siddhartha Sen, *Microsoft Research*

Haiying Shen, *University of Virginia*

Alex Shraer, *Apple*

Keith A. Smith, *NetApp*

Brent Stephens, *University of Wisconsin-Madison*

Christopher Stewart, *Ohio State University*

Patrick Stuedi, *IBM Research*

Swaminathan Sundararaman, *ParallelM*

Lalith Suresh, *VMware Research*

Bhuvan Urgaonkar, *Pennsylvania State University*

Peter Varman, *Rice University*

Shivaram Venkataraman, *University of Wisconsin-Madison and Microsoft Research*

Andrew Wang, *Cloudera*

Youjip Won, *Hanyang University*

Gala Yadgar, *Technion Israel Institute of Technology*

Antonia Zhai, *University of Minnesota*

Irene Zhang, *Microsoft Research*

Yiying Zhang, *Purdue University*

Zheng (Eddy) Zhang, *Rutgers University*

External Reviewers

Dennis Appelt
Owen Arden
Alexandre Bartel
Ketan Bhardwaj
Christian Cachin
Haibo Chen
Heming Cui
Ariel Feldman
Cristiano Giuffrida
Arjun Guha
Wajih Ul Hassan
Henry Hoffmann
Cheol-Ho Hong
Jeff Huang
Yeongji n Jang
Mehdi Javanmard
Manos Kapritsos
Hokeun Kim
Byoungyoung Lee
Sangho Lee
Sinisa Matetic
Derek Murray
Kartik Nayak
Nohhyun Park
Mathias Payer
Amir Rahmati
Nuno Santos
Marco Serafini
Zechao Shang
Abhi Shelat
Philip Shilane
Vijay Sivaraman
Chengyu Song
Duong Ta
Rebecca Taft
Chia-Che Tsai
Balajee Vamanan
Marko Vukolic
Michael Wei
Yingjun Wu
Mark Yampolskiy
Andreas Zeller
Wenchao Zhou

Message from the 2018 USENIX Annual Technical Conference Program Co-Chairs

Welcome to the 2018 USENIX Annual Technical Conference.

We are excited for ATC '18. We have had some very high-quality submissions, and clearly there has been a lot of work on behalf of both the authors who have contributed content and reviewers who have thoroughly reviewed submissions. In particular, we want to thank the program committee members and external reviewers who were willing to volunteer their time and also willing to take on a larger-than-expected load to ensure proper reviewing.

The incredible dedication by this year's program committee resulted in a program of 76 refereed papers and one keynote. These papers and keynote present novel research contributions and practical insights that advance the state-of-the-art in systems from a wide range of perspectives, demonstrating new capabilities or improvements for a variety of platforms and application scenarios. Given the spectrum of topics covered in the program, you are likely to find interesting ideas addressing your favorite areas and challenges.

For the traditional refereed papers track, we received a record number of paper registrations and submissions this year. Authors registered 557 papers, of which 377 (a 33% increase over last year) were complete submissions. The program co-chairs rejected one paper up front due to serious formatting violations. Of the submitted papers, 30 were short papers, which had to be at most five pages long (plus references), and the other 347 were full-length papers, which had to be at most 11 pages long plus references.

We required authors to submit abstracts a week before the paper submission in the hope of ensuring proper subject area coverage by the program committee and to get an idea of the reviewing load. This did not work. We had over 550 submitted abstracts, meaning almost 40% of the submissions were abandoned. In the end, requiring abstracts to be submitted early did not help with planning due to such a large number of abstracts that did not result in a submission.

The program committee had 72 members, excluding the 2 co-chairs. 28 of them had affiliations with industrial organizations, 42 with academic organizations (one member had dual affiliations), and 2 with government lab organizations. The committee represented three continents and seven countries. Program committee members were allowed to submit papers. The program co-chairs did not submit any papers.

We followed standard rules for handling conflicts of interest: conflicted members (or co-chairs) left the room during the discussion of conflicted papers. We followed the tradition of single-blind reviews. It was not a decision we explicitly made. Given the issues surrounding single-blind reviews, it was a decision that we should have thought about and justified. Reviews were done by the program committee in two rounds with a few external reviews. The chairs did not participate in any of the reviews. In the first round, each of the 377 submitted papers received at least two reviews. 219 (58%) of the papers moved to the second round. Each paper in round two received at least two additional reviews.

One of our goals was to get a large enough program committee so that we would not overwhelm members with review load and to give members enough time to produce quality reviews. Lower reviews also enable access to more potential committee members. We also had a light (20 members) and heavy committee (52). The light committee had an expected load of 12 to 16 papers and were not expected to attend the in-person committee meeting. The heavy committee had an expected load of 14-18 papers and were expected to attend the in-person meeting. As the load is relatively similar between light and heavy PC, we do not distinguish them on the website. When we saw the higher than expected number of submissions, we grew the committee. Furthermore, for papers where we lacked reviewer expertise in the main PC, we solicited 43 external reviewers, each reviewing on average one paper. In the end our committee members reviewed the maximum expected number of papers with a few going over the expectation. Altogether, we had more than 1,230 reviews.

After two phases of reviews, an online discussion was conducted among reviewers, during which the program committee decided to pre-accept 26 highly-ranked papers and pre-reject 69 more papers. These papers were not discussed in the PC Meeting while the rest of the round 2 papers, 124 papers, were discussed during the in-person program committee meeting.

The PC meeting was held on April 16–17 at the Facebook campus in Menlo Park, CA; more than 50 PC members attended the meeting in person and many others called in. During the meeting, 50 additional papers were accepted. Among these 76 acceptances, four were short papers. Because of the large number of papers to discuss, we had two parallel meetings run by each chair. We would like to thank PhD students Huaicheng Li and Mingzhe Hao of the University of Chicago for optimizing the scheduling of discussions for the meetings and for acting as scribes during the meetings. We also would like to thank Facebook engineers and staff for helping with the logistics of the PC meeting.

We added to the program one keynote, chosen from recommendations made by members of the program committee. We were not able to have any additional sessions due to the large number of presentations for accepted papers.

We are very grateful to all who contributed to ATC '18. In addition to the authors who submitted their work for consideration, the program committee, and the external reviewers, we would like to thank the USENIX staff for their outstanding conference management. By taking care of all organizational details, they enabled us to focus on building a strong program. We would also like to thank Facebook for their generosity in hosting the PC meeting.

We hope that you enjoy the conference. Thank you for participating in the USENIX ATC community!

USENIX ATC '18 Program Co-Chairs
Haryadi Gunawi, *University of Chicago*
Benjamin Reed, *Facebook*

USENIX ATC ’18:
2018 USENIX Annual Technical Conference

July 11–13, 2018
Boston, MA, USA

Performance

Kernel

Security 1

Virtualization

Security 2

Multicore

(continued on next page)

Problem Determination

Consistency

Big Data Faster

Availability

Big Data 1

Analyzing Code

Big Data 2

(continued on next page)

SSDs

The Network

Storage 1

Transactions

Storage 2

Data Center/Machine Learning

Key/Value Storage

Tributary: spot-dancing for elastic services with latency SLOs

Aaron Harlap[§*] Andrew Chung[§*] Alexey Tumanov[†]
Gregory R. Ganger[§] Phillip B. Gibbons[§]
[§]*Carnegie Mellon University* [†]*UC Berkeley*

Abstract

The Tributary elastic control system embraces the uncertain nature of transient cloud resources, such as AWS spot instances, to manage elastic services with latency SLOs more robustly and more cost-effectively. Such resources are available at lower cost, but with the proviso that they can be preempted *en masse*, making them risky to rely upon for business-critical services. Tributary creates models of preemption likelihood and exploits the partial independence among different resource offerings, selecting collections of resource allocations that satisfy SLO requirements and adjusting them over time, as client workloads change. Although Tributary's collections are often larger than required in the absence of preemptions, they are cheaper because of both lower spot costs and partial refunds for preempted resources. At the same time, the often-larger sets allow unexpected workload bursts to be absorbed without SLO violation. Over a range of web service workloads, we find that Tributary reduces cost for achieving a given SLO by 81–86% compared to traditional scaling on non-preemptible resources, and by 47–62% compared to the high-risk approach of the same scaling with spot resources.

1 Introduction

Elastic web services have been a cloud computing staple from the beginning, adaptively scaling the number of machines used over time based on time-varying client workloads. Generally, an adaptive scaling policy seeks to use just the number of machines required to achieve its *Service Level Objectives (SLOs)*, which are commonly focused on response latency and ensuring that a given percentage (*e.g.*, 95%) of requests are responded to in under a given amount of time [17, 28, 19]. Too many machines results in unnecessary cost, and too few results in excess customer dissatisfaction. As such, much research and development has focused on doing this well [20, 14, 11, 12, 26].

Elastic service scaling schemes generally assume independent and infrequent failures, which is a relatively safe assumption for high-priority allocations in private clouds and *non-preemptible* allocations in public clouds (e.g., on-demand instances in AWS EC2 [3]). This assumption enables scaling schemes to focus on client workload and server responsiveness variations in determining changes to the number of machines needed to meet SLOs.

Modern clouds also offer transient, *preemptible* resources (e.g., EC2 Spot Instances [1]) at a discount of 70–80% [6], creating an opportunity for cheaper service deployments. But, simply using standard scaling schemes fails to address the risks associated with such resources. Namely, preemptions should be expected to be more frequent than failures and, more importantly, preemptions often occur in bulk. Akin to co-occurring failures, bulk preemptions can cause traditional scaling schemes to have sizable gaps in SLO attainment.

This paper describes Tributary, a new elastic control system that exploits transient, preemptible resources to reduce cost and increase robustness to unexpected workload bursts. Tributary explicitly recognizes the bulk preemption risk, and it exploits the fact that preemptions are often not highly correlated across different pools of resources in heterogeneous clouds. For example, in AWS EC2, there is a separate spot market for each instance type in each availability zone, and researchers have noted that they often move independently: while preemptions within each spot market are correlated, across spot markets they are not [16]. To safely use preemptible resources, Tributary acquires collections of resources drawn from multiple pools, modified as resource prices change and preemptions occur, while endeavoring to ensure that no single bulk preemption would cause SLO violation. We refer to this dynamic use of multiple preemptible resource pools as *spot-dancing*.

AcquireMgr is Tributary's component that decides the resource collection's makeup. It works with any traditional scaling policy that determines (reactively or predictively) how many cores or machines are needed for each successive period of time, based on client load variation. AcquireMgr decides *which* instances will provide sufficient likelihood of meeting each time period's target at the lowest expected cost. Its probabilistic algorithm combines resource cost and preemption probability predictions for each pool to decide how many resources to include from each pool, and at what price to bid for any new resources (relative to the current market price).

[*] Equal contribution

Given that a preemption occurs when a market's spot price exceeds the bid price given at resource acquisition time, AcquireMgr can affect the preemption probability via the delta between its bid price and the current price, informed by historical pricing trends. In our implementation, which is specialized to AWS EC2, the predictions use machine learning (ML) models trained on historical EC2 Spot Price data. The expected cost of the computation takes into account EC2's policy of partial refunds for preempted instances, which often results in AcquireMgr choosing high-risk instances and achieving even bigger savings than just the discount for preemptibility.

In addition to the expected cost savings, Tributary's spot-dancing provides a burst tolerance benefit. Any elastic control scheme has some reaction delay between an unexpected burst and any resulting addition of resources, which can cause SLO violations. Because Tributary's resource collection is almost always bigger than the scaling policy's most recent target in order to accommodate bulk preemptions, extra resources are often available to handle unexpected bursts. Of course, traditional elastic control schemes can also acquire extra resources as a buffer against bursts, but only at a cost, whereas the extra resources when using Tributary are a bonus side-effect of AcquireMgr's robust cost savings scheme.

Results for four real-world web request arrival traces and real AWS EC2 spot market data demonstrate Tributary's cost savings and SLO benefits. For each of three popular scaling policies (one reactive and two predictive), Tributary's exploitation of AWS spot instances reduces cost by 81–86% compared to traditional scaling with on-demand instances for achieving a given SLO (e.g., 95% of requests below 1 second). Compared to unsafely using traditional scaling with spot instances (*AWS AutoScale* [2]) instead of on-demand instances, Tributary reduces cost by 47–62% for achieving a given SLO. Compared to other recent systems' policies for exploiting spot instances to reduce cost [24, 16], Tributary provides higher SLO attainment at significantly lower cost.

This paper makes four primary contributions. First, it describes Tributary, the first resource acquisition system that takes advantage of preemptible cloud resources for elastic services with latency SLOs. Second, it introduces AcquireMgr algorithms for composing resource collections of preemptible resources cost-effectively, exploiting the partial refund model of EC2's spot markets. Third, it introduces a new preemption prediction approach that our experiments with EC2 spot market price traces show is significantly more accurate than previous preemption predictors. Fourth, we show that Tributary's approach yields significant cost savings and robustness benefits relative to other state-of-the-art approaches.

2 Background and Related Work

Elastic services dynamically acquire and release machine resources to adapt to time-varying client load. We distinguish two aspects of elastic control, the *scaling policy* and the *resource acquisition scheme*. The scaling policy determines, at any point in time, *how many* resources the service needs in order to satisfy a given SLO. The resource acquisition scheme determines *which* resources should be allocated and, in some cases, aspects of how (e.g., bid price or priority level). This section discusses AWS EC2 spot instances and resource acquisition strategies to put Tributary and its new approach to resource acquisition into context.

2.1 Preemptible resources in AWS EC2

In addition to non-preemptible, or *reliable* resources, most cloud infrastructures offer preemptible resources as a way to increase utilization in their datacenters. Preemptible resources are made available, on a best-effort basis, at decreased cost (in for-pay settings) and/or at lower priority (in private settings). This subsection describes preemptible resources in AWS EC2, both to provide a concrete example and because Tributary and most related work specialize to EC2 behavior.

EC2 offers "on-demand instances", which are reliable VMs billed at a flat per-second rate. EC2 also offers the same VM types as "spot instances", which are preemptible but are usually billed at prices significantly lower (70% - 80%) than the corresponding on-demand price. EC2 may preempt spot instances at any time, thus presenting users with a trade-off between reliability (on-demand) and cost savings (spot).

There are several properties of the AWS EC2 spot market behavior that affect customer cost savings and the likelihood of instance preemption. (1) Each instance type in each availability zone has a unique AWS-controlled spot market associated with it, and AWS's spot markets are not truly free markets [9]. (2) Price movements among spot markets are not always correlated, even for the same instance type in a given region [23]. (3) Customers specify a bid in order to acquire a spot instance. The bid is the maximum price a customer is willing to pay for an instance in a specific spot market; once a bid is accepted by AWS, it cannot be modified. (4) A customer is billed the spot market price (not the bid price) for as long as the spot market price for the instance does not exceed the bid price or until the customer releases it voluntarily. (5) As of Oct 2nd, 2017, AWS charges for the usage of an EC2 instance up to the second, with one exception: if the spot market price of an instance exceeds the bid price during its first hour, the customer is refunded fully for its usage. No refund is given if the spot instance is revoked in any subsequent hour. We define the period where preemption makes the instance free as the *preemption window*.

When using EC2 spot instances, the bidding strategy plays an important role in both cost and preemption probability. Many bidding strategies for EC2 spot instances have been studied [9, 33, 30]. The most popular strategy by far is to bid the on-demand price to minimize the odds of preemption [23, 21], since AWS charges the market price rather than the bid price.

2.2 Cloud Resource Acquisition Schemes

Given a target resource count from a scaling policy, a resource acquisition scheme decides *which* resources to acquire based on attributes of resources (e.g., bid price or priority level). Many elastic control systems assume that all available resources are equivalent, such as would be true in a homogeneous cluster, which makes the acquisition scheme trivial. But, some others address resource selection and bidding strategy aspects of multiple available options. Tributary's AcquireMgr employs novel resource acquisition algorithms, and we discuss related work here.

AWS AutoScale [2] is a service provided by AWS that maintains the resource footprint according to the target determined by a scaling policy. At initialization time, if using on-demand instances, the user specifies an instance type and availability zone. Whenever the scaling target changes, AutoScale acquires or releases instances to reach the new target. If using spot instances, the user can use a so-called "spot fleet"[4] consisting of multiple instance type and availability zone options. In this case, the user configures AutoScale to use one of two strategies. The *lowestPrice* strategy will always select cheapest current spot price of the specified options. The *diversified* strategy will use an equal number of instances from each option. Tributary bids aggressively and diversifies based on predicted preemption rates and observed inter-market correlation, resulting in both higher SLO attainment and lower cost than AutoScale.

Kingfisher [26] uses a cost-aware resource acquisition scheme based on using integer linear programming to determine a service's resource footprint among a heterogeneous set of non-preemptible instances with fixed prices. Tributary also selects from among heterogeneous options, but addresses the additional challenges and opportunities introduced by embracing preemptible transient resources. Several works have explored ways of selecting and using spot instances. HotSpot [27] is a resource container that allows an application to suspend and automatically migrate to the most cost-efficient spot instance. While HotSpot works for single-instance applications, it is not suitable for elastic services since its migrations are not coordinated and it does not address bulk preemptions.

SpotCheck [25] proposes two methods of selecting spot markets to acquire instances in while always bidding at a configurable multiple of the spot instance's corresponding on-demand price. The first method is *greedy cheapest-first*, which picks the cheapest spot market. The second method is *stability-first*, which chooses the most price-stable market based on past market price movement. SpotCheck relies on VM migration and hot spares (on-demand or otherwise) to address revocations, which incurs additional cost, while Tributary uses a diverse pool of spot instances to mitigate revocation risk.

BOSS [32] hosts key-value stores on spot instances by exploiting price differences across pools in different data-centers and creating an online algorithm to dynamically size pools within a constant bound of optimality. Tributary also constructs its resource footprint from different pools, within and possibly across data-centers. Whereas BOSS assumes non-changing storage capacity requirements, Tributary dynamically scales its resource footprint to maintain the specified latency SLO while adapting to changes in client workload.

Wang et al. [31] explore strategies to decide whether, in the face of changing application behavior, it is better to reserve discounted resources over longer periods or lease resources at normal rates on a shorter term basis. Their solution combines on-demand and "reserved" (long term rental at discount price) instances, neither of which are ever preempted by Amazon.

ExoSphere [24] is a virtual cluster framework for spot instances. Its instance acquisition scheme is based on market portfolio theory, relying on a specified risk averseness parameter (α). ExoSphere formulates the *return* of a spot instance acquisition as the difference between the on-demand cost and the expected cost based on past spot market prices. It then tries to maximize the return of a set of instance allocations with respect to risk, considering market correlations and α, determining the fraction of desired resources to allocate in each spot market being considered. For a given virtual cluster size, ExoSphere will acquire the corresponding number of instances from each market at the on-demand price. Unsurprisingly, since it was created for a different usage model, ExoSphere's scheme is not a great fit for elastic services with latency SLOs. We implement ExoSphere's scheme and show in Section 5.6 that Tributary achieves lower cost, because it bids aggressively (resulting in more preemptions), and higher SLO attainment, because it explicitly predicts preemptions and selects resource sets based on sufficient tolerance of bulk preemptions.

Proteus [16] is an elastic ML system that combines on-demand resources with aggressive bidding of spot resources to complete batch ML training jobs faster and cheaper. Rather than bidding the on-demand price, it bids close to market price and aggressively selects spot markets and bid prices that it predicts will result in preemption, in hopes of getting many partial hours of free resources. The few on-demand resources are used to maintain a copy of the dynamic state as spot instances come and go, and acquisitions are made and used to scale the parallel computation whenever they would reduce the average cost per unit work. Although Tributary uses some of the same mindset (aggressive use of preemptible resources), elastic services with latency SLOs are different than batch processing jobs; elastic services have a target resource quantity for each point in time, and having fewer usually leads to SLO violations, while having more often provides no benefit. Unsurprisingly, therefore, we find that Proteus's scheme is not a great fit for such services. We implement Proteus's acquisition scheme and show in Section 5.6 that Tributary achieves much higher SLO attainment, because it understands the resource target and explicitly uses diversity to mitigate bulk preemption effects. Tributary also uses a new and much more accurate preemption predictor.

3 Elastic Control in Tributary

AcquireMgr is Tributary's resource acquisition component, and its approach differentiates Tributary from previous elastic control systems. It is coupled with a scaling policy, any of many popular options, which provides the time-varying resource quantity target based on client load. AcquireMgr uses ML models to predict the preemption probability of resources and exploits the relative independence of AWS spot markets to account for potential bulk preemptions by acquiring a diverse mix of preemptible resources collectively expected to satisfy the user-specified latency SLO. This section describes how AcquireMgr composes the resource mix while targeting minimal cost.

Resource Acquisition. AcquireMgr interacts with AWS to request and acquire resources. To do so, AcquireMgr builds sets of request vectors. Each request vector specifies the instance type, availability zone, bid price, and number of instances to acquire. We call this an *allocation request*. An *allocation* is defined as a set of instances of the same type acquired at the same time and price. AcquireMgr's *total footprint*, denoted with the variable A, is a set of such allocations. Resource acquisition decisions are made under four conditions: (1) a periodic (one-minute) clock event fires, (2) an allocation reaches the end of its preemption window, (3) the scaling policy specifies a change in resource requirement, and/or (4) a preemption occurs. We term these conditions *decision points*.

AcquireMgr abstracts away the resource type which is being optimized for. For the workloads described in this paper, virtual CPUs (VCPUs) are the bottleneck resource; however, it is possible to optimize for memory, network bandwidth, or other resource types instead. A service using Tributary provides its resource scaling characteristics to AcquireMgr in the form of a *utility function* $\upsilon()$. This *utility function* maps the number of resources to the percentage of requests expected to meet the target latency, given the load on the web service. The shape of a utility function is service-specific and depends on how the service scales, for the expected load, with respect to the number of resources. In the simplest case where the web service is embarrassingly parallel, the utility function is linear with respect to the number of resources offered until 100% of the requests are expected to be satisfied, at which point the function turns into a horizontal line. As a concrete example, if an embarrassingly parallel service specifies that 100 instances are required to handle 10000 requests per second without any of the requests missing the target latency, a linear utility function will assume that 50 instances will allow the system to meet the target latency on 50% of the requests. Tributary allows applications to customize the utility function so as to accommodate the resource requirements of applications with various scaling characteristics.

In addition to providing $\upsilon()$, the service also provides the application's target SLO in terms of a percentage of requests required to meet the target latency. By exposing the target SLO as a customizable input, Tributary allows the application to control the Cost-SLO tradeoff. Upon receiving this information, AcquireMgr acquires enough resources to meet SLO in expectation while optimizing for expected cost. In deciding which resources to acquire, AcquireMgr uses the prediction models described in Sec. 3.1 to predict the probability that each allocation would be preempted. Using these predictions, AcquireMgr can compute the expected cost and the expected utility of a set of allocations (Sec. 3.2). AcquireMgr greedily acquires allocations until the expected utility is greater than or equal to the SLO percentage requirement (Sec. 3.3).

3.1 Prediction Models

When acquiring spot instances on AWS, there are three configurable parameters that affect preemption probability: instance type, availability zone and bid price. This section describes the models used by AcquireMgr to predict allocation preemption probabilities.

Previous work [16] proposed taking the historical median probability of preemption based on the instance type, availability zone and bid price. This approach does not consider time of day, day of week, price fluctuations and several other factors that affect preemption probabilities. AcquireMgr trains ML models considering such features to predict resource reliability.

Training Data and Feature Engineering. The prediction models are trained ahead of time with data derived from AWS spot market price histories. Each sample in the training dataset is a *hypothetical bid*, and the *target variable*, `preempted`, of our model is whether or not an instance acquired with the hypothetical bid is preempted before the end of its preemption window (1 hr). We use the following method to generate our data set: For each instance and *bid delta* (bid price above the market price with range $[0.00001, 0.2]$) we generate a set of hypothetical bids with the bid starting at a random point in the spot market history. For each bid, we look forward in the spot market price history. If the market price of the instance rises above the bid price at any point within the hour, we mark the sample as `preempted`. For each historical bid, we also record the ten prices immediately prior to the random starting point and their time-stamps.

To increase prediction accuracy, AcquireMgr engineers features from AWS spot market price histories. Our engineered features include: (1) Spot market price; (2) Average spot market price; (3) Bid delta; (4) Frequency of spot market price changing within past hour; (5) Magnitude of spot market price fluctuations within past two, ten, and thirty minutes; (6) Day of the week; (7) Time of day; (8) Whether the time of day falls within working hours (separate feature for all three time zones). These features allow AcquireMgr to construct a more complex prediction model, leading to a higher prediction accuracy (Sec. 5.7).

Model Design. To capture the temporal nature of the EC2 spot market, AcquireMgr uses a *Long Short-Term Memory Recurrent Neural Network (LSTM RNN)* to predict instance preemptions. The LSTM RNN is a popular model for workloads where the ordering of training examples is important to prediction accuracy [29]. Examples of such workloads include language modeling, machine translation, and stock market prediction. Unlike feed forward neural networks, LSTM models take previous inputs into account when classifying input data. Modeling the EC2 spot market as a *sequence* of events significantly improves prediction accuracy (Sec. 5.7). The output of the model is the probability of the resource being preempted within the hour.

3.2 AcquireMgr

To make decisions about which resources to acquire or release, AcquireMgr computes the expected cost and expected utility of the set of instances it is considering at each decision point. Calculations of the expected values are based on probabilities of preemption computed by AcquireMgr's trained LSTM model. This section describes how AcquireMgr computes these values.

Definitions. To aid in discussion, we first define the notion of a *resource pool*. Each instance type in each availability zone forms its own *resource pool*—in the context of the EC2 spot instances, each such resource pool has its own spot market. Given a *set of allocations* A, where A is formulated as a jagged array, let A_i be defined as the i^{th} entry of A corresponding to an array of allocations from resource pool i sorted by bid price in ascending order. We define allocation $a_{i,j}$ as an allocation from resource pool i (*i.e.*, $a_{i,j} \in A_i$) with the j^{th} lowest bid in that resource pool. We further denote $p_{i,j}$ as the bid price of allocation $a_{i,j}$, $\beta_{i,j}$ as the probability of preemption of allocation $a_{i,j}$, and $t_{i,j}$ as the time remaining in the preemption window for allocation $a_{i,j}$. Note that $p_{i,j} \geq p_{i,j-1}$, which also implies $\beta_{i,j-1} \geq \beta_{i,j}$. Finally, we define a *size*(A) function that returns the size of A's major dimension. See Table 1 for symbol reference.

A	Set of allocations as jagged array
A_i	Sorted array of allocations from resource pool i
$a_{i,j}$	Set of instances allocated from resource pool i
$\beta_{i,j}$	Probability that allocation $a_{i,j}$ is preempted
$t_{i,j}$	Time left in the preemption window for $a_{i,j}$
$k_{i,j}$	Number of instances in allocation $a_{i,j}$
$P_{i,j}$	Market price of allocation $a_{i,j}$
$p_{i,j}$	Bid price of allocation $a_{i,j}$
$size(y)$	Size of the major dimension of array y
$resc(y)$	Counts the total number of resources in y
λ_i	Regularization term for diversity
$P(R=r)$	Probability that r resources remain in A
$\upsilon(r)$	The utility of having r resources remain in A
V_A	The expected utility of a set of allocations A
C_A	Expected cost of a set of allocations ($)

Table 1: Summary of parameters used by AcquireMgr

Expected Cost. The total expected cost for a given footprint A is calculated as the sum over the expected cost of individual allocations $C_A[a_{i,j}]$:

$$C_A = \sum_{i=1}^{size(A)} \sum_{j=1}^{size(A_i)} C_A[a_{i,j}] \tag{1}$$

AcquireMgr calculates the *expected* cost of an allocation by considering the probability of preemption within the preemption window $\beta_{i,j}$ for a given allocation $a_{i,j}$ at a given *bid delta*. There are exactly two possibilities: an allocation will either be preempted with probability $\beta_{i,j}$ or it will reach the end of its preemption window in the remaining $t_{i,j}$ minutes with probability $1-\beta_{i,j}$, in which case we would voluntarily release the allocation. The expected cost can then be written down as:

$$C_A[a_{i,j}] = (1-\beta_{i,j}) * P_{i,j} * k_{i,j} * t_{i,j} + \beta_{i,j} * 0 * k_{i,j} * t_{i,j} \tag{2}$$

where $k_{i,j}$ is the number of instances in the allocation. and $P_{i,j}$ is the market price for instance of type i at the time of acquisition.

Expected Utility. In addition to computing expected cost for a set of allocations, AcquireMgr computes the *expected utility* for a set of allocations. The *expected utility* is the expected percentage of requests that will meet the latency target given the set of allocations A. Expected utility takes into account the probability of allocation preemptions, providing AcquireMgr with a metric for quantifying the expected contribution that each allocation should make to meet the resource target. The expected utility V_A of the set of allocations A is calculated as follows:

$$V_A = \sum_{r=0}^{resc(A)} P(R=r) * \upsilon(r) \tag{3}$$

where $P(R)$ is the probability mass function of the discrete random variable R that denotes the number of resources not preempted within the next hour, υ is the utility function provided by the service, and $resc(A)$ is the function that reports the number of resources in a set of allocations A. $resc(A)$ computes the *total amount of resources* in A, while $size(A)$ only computes the *size* of A's major dimension.

Eq. 3 computes the expected utility over the next hour given a workload, as though Tributary just bid for all its allocations. This works, even though there will usually be complex overlapping expiration windows across an hour, because it only needs to hold true until recomputed at the next decision point, which is never more than a minute away. To derive $P(R)$, AcquireMgr starts off with the original set of allocations A and generates all possible subsets of A. Each possible subset $S \subseteq A$, S marks some allocations in A as preempted ($\in S$) and the remaining allocations as not preempted ($\notin S$). To formalize the notion, we define the indicator variable $d_{i,j}$, where $d_{i,j} = 1$ if allocation $a_{i,j} \in S$ and $d_{i,j} = 0$ otherwise.

To compute the probability of S being the set of preempted resources ($P(S)$), AcquireMgr separates all allocations by resource pools, as each resource pool within AWS has its own spot market. We leverage the following localizing property. Within each resource pool A_i, the probability of preempting an allocation $a_{i,j}$ is only dependent on whether the allocation with the next lowest bid price, $a_{i,j-1}$, in the same resource pool is preempted. Note that $P(a_{i,1}) = \beta_{i,1}$ for allocation $a_{i,1}$ for all resource pools i. Consider two allocations $a_{i,j}, a_{i,j-1} \in A$ from resource pool A_i. We observe the following properties: (1) $a_{i,j}$ cannot be preempted unless $a_{i,j-1}$ is preempted, (2) the probability that both $a_{i,j}$ and $a_{i,j-1}$ are preempted is the probability that $a_{i,j}$ is preempted, and (3) the probability that $a_{i,j}$ is preempted without $a_{i,j-1}$ being preempted is 0. With Bayes' Rule, we observe that:

$$P(a_{i,j}|a_{i,j-1}) = \frac{P(a_{i,j} \wedge a_{i,j-1})}{P(a_{i,j-1})} = \frac{\beta_{i,j}}{\beta_{i,j-1}}. \tag{4}$$

Thus, for an allocation $a_{i,j}$ given subset $S \subseteq A$,

$$P(a_{i,j}|a_{i,j-1}) = \begin{cases} 0 & \text{if allocation } a_{i,j-1} \notin S, \\ \beta_{i,j}/\beta_{i,j-1} & \text{else.} \end{cases} \tag{5}$$

Tributary further introduces a regularization term λ_i to encourage bidding in markets with low correlation. Having instances spread across lowly correlated markets is important for avoiding high-risk footprints. If the resource footprint has too many instances from correlated resource pools, Tributary becomes exposed to having too many resources being lost to a correlated price spike, potentially causing an SLO violation. In order obtain price correlation across spot markets, we periodically keep track of fix-sized moving windows of spot markets and compute the Pearson correlation between each pair of spot markets. When computing *expected utility*, Tributary increases an allocation in A_i's probability of preemption $\beta_{i,j}$ by λ_i:

$$\lambda_i = \gamma * \sum_{l=1}^{size(A)} \rho_{i,l} * \frac{resc(A_i) + resc(A_l)}{2 * resc(A)} \tag{6}$$

where $\rho_{i,l}$ is the Pearson correlation between resource pools i and l, and $\gamma \in \mathbb{R} \geq 0$ is the configurable penalty multiplier. Essentially, we add a weighted penalty to an allocation based on its Pearson correlation scores with the rest of our resources in different resource pools. In our experiments, we set $\gamma = 0.01$. The regularization term leads to Tributary creating a diversified resource pool, thus reducing the probability that a significant portion of the resources are preempted simultaneously. Having a high probability of maintaining the majority of the resource pool at any point time, allows Tributary to avoid SLO violations with a high probability.

Let's denote $P(S)$ as the probability of S being the set of resources preempted from A. AcquireMgr computes it by taking the product of the conditional probability of each allocation having the outcome specified in S. If the allocation is preempted ($d_{i,j} = 1$) the conditional probability of the allocation being preempted ($P(a_{i,j}|a_{i,j-1})$) is used, otherwise ($d_{i,j} = 0$) the product uses the conditional probability of the allocation not being preempted $(1 - P(a_{i,j}|a_{i,j-1}))$.

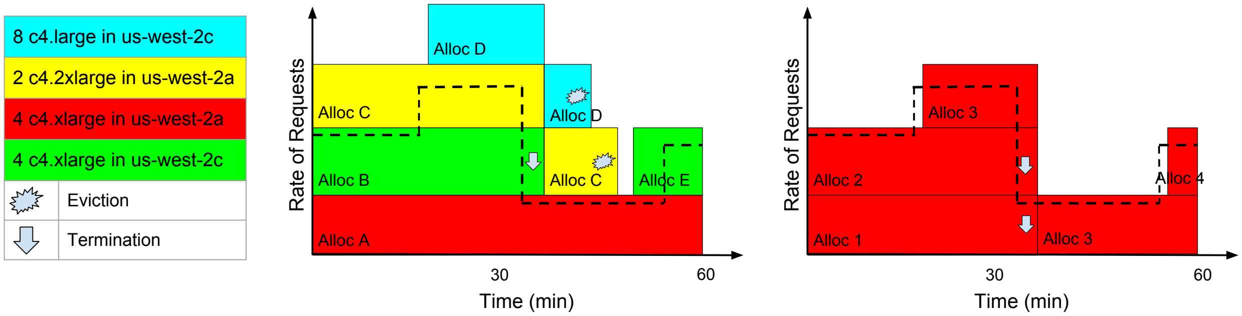
$$P(S) = \prod_{i=1}^{size(A)} \prod_{j=1}^{size(A_i)} \Big(d_{i,j} * P(a_{i,j}|a_{i,j-1}) + (1 - d_{i,j}) * (1 - P(a_{i,j}|a_{i,j-1})) \Big) \tag{7}$$

Finally, AcquireMgr formulates the probability of r resources remaining after preemption $P(R = r)$ (Eq. 3) as the sum of the probabilities of all sets S where the number of resources not preempted in S equals to r:

$$P(R=r) = \sum_{S \subseteq A, resc(S) = resc(A) - r} P(S) \tag{8}$$

which it uses to calculate the expected utility of a set of allocations A (Eq. 3).

Computational tractability. AcquireMgr's algorithm is exponentially computationally expensive as the number of spot markets considered increases. When considering more markets, it is possible to reduce computational complexity by grouping similar, correlated spot



(a) Legend (b) Tributary (c) AutoScale

Figure 1: Figures (b) and (c) show how Tributary and AutoScale handle a sample workload respectively. Figure (a) is the legend for (b) and (c), color-coding each allocation. The black dotted lines in (b) and (c) signify the request rates over time. At **minute 15**, the request rate unexpectedly spikes and AutoScale experiences “slow” requests until completing integration of additional resources with *3*. Tributary, meanwhile, had extra resources meant to address preemption risk in *C*, providing a natural buffer of resources that is able to avoid “slow” requests during the spike. At **minute 35**, when the request rate decreases, Tributary terminates *B*, since it believes that *B* has the lowest probability of getting the free partial hour. It does not terminate *D* since it has a high probability of eviction and is likely to be free; it also does not terminate *C* since it needs to maintain resources. AutoScale, on the other hand, terminates both *2* and *3*, incurring partial cost. At **minute 52**, the request rate increases and Tributary again benefits from the extra buffer while AutoScale misses its latency SLO. In this example, Tributary has less “slow” requests and achieves lower cost than AutoScale because AutoScale pays for *3* and for the partial hour for both *1* and *2* while Tributary only pays for *A* and the partial hour for *B* since *C* and *D* were preempted and incur no cost.

markets, and performing revocation analysis with a representative market. Although this would potentially decrease the precision of the preemption analysis, it would allow AcquireMgr to further improve performance by considering a larger number of markets.

3.3 Scaling Out

Resource Acquisition. When Tributary starts, the user specifies a *target SLO* in terms of percentage of requests that respond within a certain latency for Tributary to target. AcquireMgr uses this target SLO to acquire resources. At each decision point, AcquireMgr’s objective is to acquire resources until the expected utility θ_A is greater than or equal to the target SLO. If the expected utility is greater than or equal to the target SLO, no action is taken; otherwise, AcquireMgr computes the expected cost (Eq. 2) and utility of the current set of allocations (Eq. 3). AcquireMgr then calculates the missing number of resources (M) required to meet the target SLO and builds a set of possible allocations (Λ) that consists of allocations from different resource pools at different bid prices (from \$0.0001 to \$0.2 above the current price). For each possible allocation Λ_i, AcquireMgr records the new expected utility divided by the new expected cost of $A \cup \Lambda_i$, choosing the allocation Λ_{chosen} that maximizes this value. AcquireMgr continues to add possible allocations until it achieves the target SLO in expectation.

Buffers of Transient Resources. To accommodate potential resource preemptions, Tributary inherently acquires more than the required amount of resources if any of its allocations have a preemption probability greater than zero, which is always the case with spot instances. The amount of additional resources acquired depends on the target SLO and the probabilities of allocation preemptions (Eq. 3). While the primary goal of these additional resources is to account for preemptions, they often have the added benefit handling unexpected increases in load. Experiments with Tributary show that these resource buffers both increase the fraction of requests meeting latency targets and decrease cost (Sec. 5.3).

3.4 Scaling In

Aside from preemptions, Tributary also tries to scale in voluntarily. As described earlier, each allocation is considered only for the duration of the preemption window. When an allocation reaches the end of its preemption window, it is terminated and replaced with a new allocation if required. When resource requirements decrease, Tributary considers terminating allocations for allocations least likely to be preempted. During this process Tributary chooses the allocation with the least time remaining in the hour, computes the expected utility θ_A without this allocation, and if it is greater than the target SLO, Tributary terminates the allocation. Tributary continues to try and terminate allocations as long as θ_A remains greater than the target SLO.

3.5 Example and Future Consideration

Example. Fig. 1 shows how Tributary and AutoScale handle a sample workload, including how the extra resources Tributary acquires to handle preemption events can also handle an unexpected request rate increase and how aggressive allocation selection can get some resources for free due to preemptions.

Future. Tributary lowers cost and meets SLO requirements by taking advantage of low-cost spot instances and uncorrelated prices across different spot instance markets. Mass adoption of systems like Tributary could change these characteristics. While a detailed analysis of mass adoption's potential effects on EC2 spot-markets is outside the scope of this paper, we evaluate the effects of two potential changes to the spot-market policies in Section 5.5.

4 Tributary Implementation

Figure 2 shows Tributary's high-level system architecture. This section describes the main components, how they fit together, and how they interact with AWS.

Preemption Prediction Models. The *prediction models* are trained offline using TensorFlow [8] and deployed using Tensorflow Serving [7]. A separate model is used for each resource pool. To service run time predictions Tributary launches a *Prediction Serving Proxy* that receives all prediction queries from AcquireMgr, forwards them to their respective models, aggregates the results, and returns the predictions to AcquireMgr.

Resource Footprint Management. In Tributary, AcquireMgr takes primary responsibility for managing the resource footprint. AcquireMgr acquires instances, terminates instances, and monitors AWS for instance preemption notifications. AcquireMgr considers modifying the resource footprint at every *decision point*, and it follows the procedure described in Sec. 3.3 when additional resources are needed. Once AcquireMgr selects a set of instances to acquire, it sends instance requests to AWS via *boto.ec2* API calls. AWS responds with a set of spot request ids, which corresponds to the EC2 instances allocated to AcquireMgr. Once the instances are in a running state, AcquireMgr sends the instance ids associated with the new instances to Resource Manager. Instance removal follows a similar procedure.

Scaling Policy. The *Scaling Policy* component determines dynamic sizing of the resource target. Through a simple event-driven API, users can implement their own scaling policies that access metrics provided by the Monitoring Manager and specify the resource target.

Monitoring Manager (MonMgr). The *Monitoring Manager* orchestrates monitoring of service system resources. The Scaling Policy can register for metrics such as total number of requests and average CPU utilization of instances. The MonMgr queries requested metrics using AWS CloudWatch each *monitoring period* and forwards them to the scaling policy.

Resource Manager (ResMgr). The *Resource Manager* is a proxy for AcquireMgr. Using resource targets provided by the Scaling Policy, the ResMgr generates the utility function used by AcquireMgr to make resource acquisition decisions.[1] The ResMgr also receives instance allocations and termination notices from AcquireMgr and forwards them to the Service Manager.

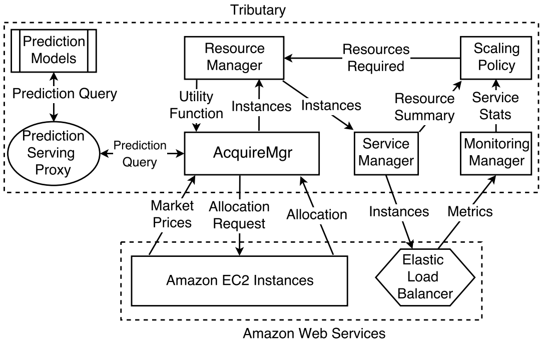


Figure 2: Tributary architecture.

5 Evaluation

This section evaluates Tributary's effectiveness. The results support a number of important findings: (1) Tributary's exploitation of AWS spot market instances reduces cost by 81%–86% compared to on-demand instances and simultaneously decrease SLO latency misses; (2) Compared to standard bidding policies for spot instances, Tributary reduces cost by up to 41% and decreases SLO latency misses by 31%–65%; (3) Compared to extending those standard policies to use enough extra (buffer) resources to match Tributary's number of SLO latency misses, Tributary reduces cost by 47%–62%; (4) Tributary outperforms state-of-the-art resource managers in running elastic services; (5) Tributary's preemption prediction models improve accuracy significantly, resulting in 37% lower cost than previous prediction approaches.

5.1 Experimental Setup

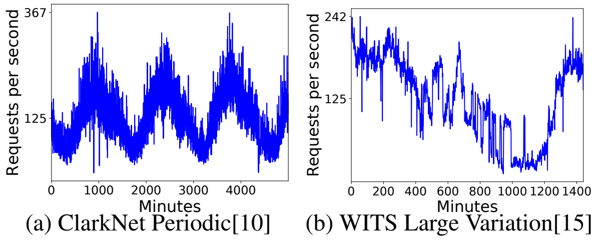
Experimental Platform. We report results for use of three AWS EC2 spot instance types: *c4.large*, *c4.xlarge*, and *c4.2xlarge*. The results correspond to the *us-west-2* region, which consists of three availability zones. Using the three instance types in each availability zone, our experiments involve nine resource pools.

Workload. The simulated workload uses a real-world trace for request arrival times, with each request consisting of the derivation of the PBKDF2 [18] key of a password. The calculation of a PBKDF2 key is CPU-heavy, with no network overhead and minimal memory overhead. With the CPU performance being the bottleneck, the resource requirement can be characterized in requests-per-second-per-VCPU.

Environment. In the simulation framework, each instance is characterized with a number of VCPUs, and the request processing time is configured to the measured time for one request on an EC2 instance ($\approx$100ms). Each instance server maintains a queue of requests, and we simulate the queueing effects using the discrete event simulation library SimPy [22]. The simulation framework takes into account resource start-up time, with newly acquired instances not able to service requests for two hundred seconds following their launch.

SLO and Scaling. The target service latency is set to

[1] Process of constructing the utility function is described in Sec. 5.2.



(a) ClarkNet Periodic[10] (b) WITS Large Variation[15]

Figure 3: Traces used in system evaluation.

one second, and we verified on EC2 that a VCPU can handle roughly 10 requests per second without violating the latency target. So, the requests-per-second-per-VCPU is ten, and the queue size per server instance is ten times the number of VCPUs in the instance. Tributary is not overly sensitive to the target latency setting.

Traces. We use four real-world request arrival traces with differing characteristics. *Berkeley* is from the Berkeley Home IP proxy service and *ClarkNet* is from the ClarkNet ISP's HTTP servers [10]. Both exhibit a periodic, diurnal pattern. We use the first 2000 minutes of these two traces, which covers an entire period. *WITS* is a sampled trace from the Waikato Internet Traffic Storage (WITS) [15]. The trace lasts for roughly a day, from April 6th to April 7th of the year 2000. This trace exhibits large variation of request rates throughout the day, as can be seen in Fig. 3b. *WorldCup98* is the arrival trace of the workload on the 1998 FIFA World Cup HTTP Servers [10] on day 75 of the World Cup. All traces are scaled to have an average of 125 requests per second in order to generate sufficient load for the experiments.

5.2 Scaling Policies Evaluated

We implement three popular scaling policies: *Reactive*, *Predictive Moving Window Average (MWA)*, and *Predictive Linear Regression (LR)* to evaluate our system. The utility function provided by the service is linear for all three policies. We make this assumption since our workload characteristic is embarrassingly parallel — if a workload exhibits different scaling characteristics, a different utility function can be employed.

The *Reactive Policy* scales out immediately when demand reported by the MonMgr is greater than what the available resources are able to handle. It scales in slowly (only after three minutes of low demand), as recommended by Gandhi et al. [12], to prevent premature scale-in in case the demand fluctuates widely in a short period of time. The *MWA Policy* maintains a sliding window of a fixed size, with each window entry consisting of the number of requests received in each monitoring period. The policy takes the average of the window entries to predict the number of requests on the next monitoring period. The policy then adjusts the utility and scaling functions according to the predicted number of requests, and reports the updated functions to the ResMgr to scale in expectation of future requests. The *LR Policy* also maintains a sliding window of a fixed size, but rather than using the average in the window for prediction, the policy performs linear regression on data points in the window to estimate the expected number of requests in the next monitoring period. Our experiments show that regardless of the scaling policy used, Tributary beats its competitors in both meeting the service latency target and cost.

5.3 Improvements with Tributary

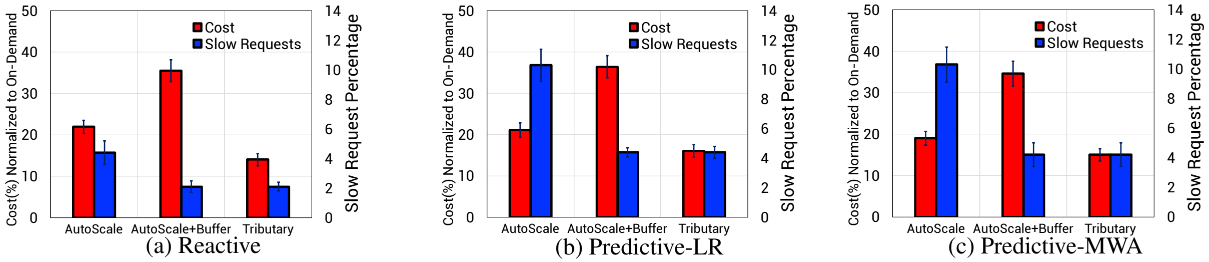
Here, we evaluate Tributary's ability to reduce cost and latency target misses against AutoScale.

AWS Autoscale. AWS AutoScale (Sec. 2.2) as offered by Amazon only supports the simplest reactive scaling policies. To provide better comparison between approaches, we implement the AWS AutoScale resource acquisition algorithm as closely as possible according to its documentation [2] and integrate it with Tributary's SvcMgr to work with its more powerful scaling policies. From here on, mentions of *AutoScale* refer to our implementation of AWS AutoScale. AutoScale is the equivalent of the AcquireMgr component of Tributary. The default AutoScale algorithm with spot instances bids for the lowest market-priced spot instance at the on-demand price upon resource requests by the scaling policy. In addition, AutoScale terminates resources as soon as the resource requirements are lowered, choosing to terminate resources that are most expensive at the moment.

Methodology and Terminology. To achieve fair comparisons across a wide range of data points, we perform cost analysis with simulations using historical spot market traces. Using traces allows us to test different approaches on the same period of market data and to get a better picture of the expected behavior of the system in a shorter amount of time. For each request arrival trace (Sec. 5.1) and resource acquisition approach, we present the average cost and percentage of *"slow" requests* over trace requests across ten randomly chosen day/time starting points between January 23, 2017 and March 23, 2017 in the *us-west-2* region. From here on, we define a *"slow" request* as a request that does not meet the latency target and the percentage of "slow" requests as the percentage of "slow" requests over all requests in a single trace. [2]

Cost Savings and Service Latency Improvements. Fig. 4 shows the cost savings and percentage of "slow" requests for the *ClarkNet* trace. The cost savings are normalized against running Tributary on on-demand resources. The results demonstrate that Tributary reduces cost and "slow" requests for all three scaling policies. Cost savings are ≈ 85% compared to on-demand resources. For the *ClarkNet* trace, Tributary reduces cost by 36%, 24% and 21% compared to to AutoScale for the *Reactive*, *Predictive-LR* and *Predictive-MWA* scaling policies, respectively. Compared to AutoScale, Tributary

[2] Prediction models were trained on data from 06/06/16 – 01/22/17.



(a) Reactive (b) Predictive-LR (c) Predictive-MWA

Figure 4: Cost savings (red) and percentage of "slow" requests (blue) for the *ClarkNet* trace.

reduces "slow" requests by 72%, 61% and 64%, respectively, for the three scaling policies.

In order to decrease the number "slow" requests, popular scaling polices are often configured to provision more resources than immediately necessary to handle unexpected increases in load. It is common to specify the resource buffer as a percentage of the expected resource requirement. For example, with a buffer of 50%, 15 resources (*e.g.*, VCPUs) would be acquired rather than the projected 10. AutoScale+Buffer shows the cost of provisioning AutoScale with a large enough buffer such that its number of "slow" requests matches that of Tributary. Tributary reduces cost by 61%, 56% and 57% compared to AutoScale+Buffer for the three scaling policies.

The cost savings for Tributary on the *Berkeley* trace relative to AutoScale are similar to those on the *ClarkNet* trace, but the reduction in percentage of "slow" requests increases. This difference in performance is due to differing characteristics of the two traces — the *ClarkNet* trace experiences more minute-to-minute volatility in request rate compared to the *Berkeley* trace. We observe similar levels of cost reductions and reduction in "slow" requests on the *WITS* and *WorldCup98* traces, results for WITS are shown in Tables 2. Compared to AutoScale+Buffer, Tributary decreased costs by 47–62% across all traces.

Scaling Policy	Cost Saving	"Slow" request Reduction
Reactive	37%	31%
Predictive-LR	33%	50%
Predictive-MWA	29%	51%

Table 2: Cost and "slow" request improvements for Tributary compared to AutoScale for the *WITS* trace

Attribution of Benefits. Tributary's superior performance arises from several factors. Much of the reduction in cost compared to AutoScale is due to Tributary's ability to get free instance hours. *Free instance hours* occur when an allocation does useful work but is preempted by AWS before the end of a preemption window. The user receives a refund for the partial hour, which means that any work done by the allocation in the preemption window comes at no cost to the user. Tributary takes the probability of getting free instance hours into account when computing the expected cost of allocations (Eq. 1), often acquiring resources that provide higher opportunities for free instance hours.

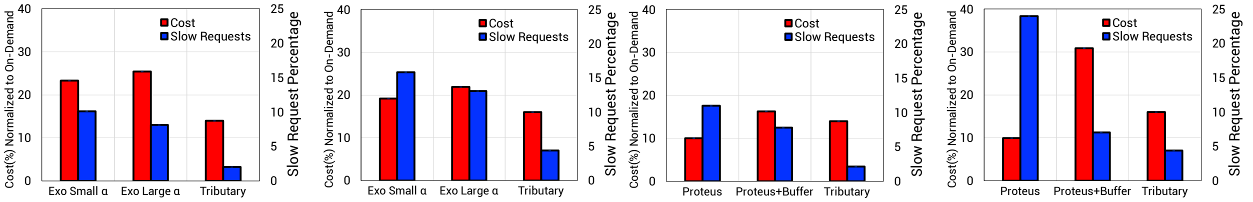
Another factor in Tributary's lower cost is its ability to remove allocations that are not likely to be preempted when demand drops. When resource demand decreases, Tributary terminates instances that are least likely to be preempted, thus lowering the expected cost of its resource footprint. The reductions in "slow" requests arise from the buffer of resources acquired by Tributary (Sec. 3.3). When acquiring instances, AcquireMgr estimates their probability of preemption. Unless all allocations have a preemption probability of zero, which never occurs for spot instances, Tributary acquires more resources than specified by the scaling policy. The primary goal of the additional resources is to ensure that, when Tributary experiences preemption events, it still has at least the specified number of resources in expectation. The additional resources also provide a secondary benefit by handling some or all of unexpected bursts of requests that exceed the load expected by the scaling policy. The cost of these additional resources is commonly offset by free instance hours; indeed, the extra resources are acquired to cope with preemptions.

5.4 Risk Mitigation

A key feature of Tributary is that it encourages instance diversification, *i.e.*, acquiring instances from mostly independent resource pools (Sec. 3.2). The default AutoScale policy is the lowest-price policy, which does not take diversification into account when acquiring instances; instead, it acquires the cheapest instance. Illustrated in Fig. 1, Tributary acquires different types of instances in different availability zones, while AutoScale acquires instances of the same type (all red). Diversifying across resource pools is important, because each has an independent spot market, avoiding highly correlated allocation preemptions within a single instance market. Acquiring too much from a single pool, as often occurs with AutoScale, creates a high risk of SLO violation when preemption events occur (*e.g.*, if the red allocation in Fig. 1c was preempted prior to minute 35).

In our experiments, we found it to be very rare for market prices to rise above on-demand prices, meaning that AutoScale rarely experiences preemption events. However, when examining past EC2 spot market traces and other availability zones, we found it to be significantly more common for the market price to rise above the on-demand price, thus preempting AutoScale instances.[3]

[3] From 01/23/17–03/20/17, the market price rose above the on-demand price 0 times for the *c4.2xlarge* instance type in *us-west-2*. From 11/1/16–01/22/17, it happened 1073 times.



(a) Reactive (b) Predictive-LR (c) Reactive (d) Predictive-LR

Figure 5: Comparing to ExoSphere and Proteus. Predictive-MWA results not shown but similar.

Since Amazon charges users the market price and not the bid price, it is possible that Amazon may once again preempt instances bidding the on-demand price with regularity—a phenomenon we recently observed in the *us-east* availability zones. Thus, AutoScale's resource acquisition approach is riskier for services with latency SLOs on spot machines.

Cost of Diversified AutoScale. In addition to the default AutoScale policy which acquires the lowest-priced instance, AWS also offers a diversified AutoScale policy that starts instances from a diverse set of resource pools [4]. Acquiring instances from different spot markets reduces preemption risks, but our experiments showed that it increases cost by 8%–12% compared to the lowest-price AutoScale policy. Compared to Tributary, which diversifies across spot markets intelligently, we found that a diversified AutoScale policy cost 68% more to achieve the same number of "slow" requests for the *reactive* scaling policy on the *ClarkNet* trace.

5.5 Pricing Model Discussion

Our experimental results are based on current AWS EC2 billing policies, as described Section 2.1. This section discusses how Tributary would function under two potential changes to the billing model: (1) elimination of preemption refunds, (2) institution of a free market.

Elimination of preemption refunds. If Amazon eliminates refunds when the market price exceeds bid price during the first hours of usage, Tributary would lose incentive to bid close to market price. Tributary's model would capture this change by setting β in Eq. 2 to zero. With higher bids, Tributary would acquire fewer resources because preemption would be less likely. The amount of resources acquired would still exceed the amount of resources required as they would still have non-zero preemption probabilities.

Although Tributary extracts significant benefit from the refunds, it still outperforms AutoScale without it. For example, in a simulation with this billing model modification, Tributary still reduces cost by 31% compared to AutoScale with sufficient buffer to match numbers of "slow" requests, for the *Clarknet* trace using the *reactive* scaling policy. As expected, Tributary continues to meet SLOs with high likelihood, as it continues to diversify its resource pool and acquire buffers of resources (albeit smaller ones) to account for preemption events.

Free market behavior. In its current design, the AWS EC2 spot markets do not behave as free markets [9]. Customers specify their bid prices for a given resource, but generally do not pay that amount. Instead, a customer is billed according to the EC2-determined spot price for that resource. It is possible, perhaps even likely as the spot market becomes widely popular, that AWS will transition toward a billing policy in which users are charged their *bid price*, instead of the *market price*, and prices move based on supply and demand rather than unknown seller policies. This change would render the commonly used strategy of bidding far above the market price (e.g., bidding the on-demand price) obsolete. Tributary's behavior would not change significantly, as it already often sets bid prices close to market prices and explicitly considers revocation risks, and we believe it would therefore outperform other approaches by even larger margins.

5.6 Comparing to State of the Art

This section compares Tributary's support for elastic services to two state-of-the-art resource managers designed for preemptible instances. Since neither system was designed for elastic services with latency SLOs, Tributary unsurprisingly performs significantly better.

Exosphere. We implemented ExoSphere's allocation strategy, described in Sec. 2.2, with the following assumptions and modifications: (i) The ExoSphere paper did not specify whether the correlation between markets is recomputed as time moves on. In order to avoid the need to constantly reconstruct ExoSphere's resource footprint, we assumed static correlation between markets. (ii) As the ExoSphere paper does not provide guidelines as to how to choose α, we experimented with a range of α from 1 to 10^9. Higher α instructs ExoSphere to be more risk averse at the expense of higher cost.

Fig. 5 shows the normalized cost and percentage of "slow" requests served for Tributary and for ExoSphere with small (1) and large (10^9) values of α. These experiments were performed on a further scaled-up version of the *ClarkNet* trace (100x of already-scaled version), since ExoSphere was designed for 100s to 1000s of instances and performs poorly at a scale of 10s.[4] In our experiments, we observed that Exosphere with a small α tends to acquire mainly the cheapest resources, inducing little diversity and increasing the number of "slow" requests in the event of preemptions. Tributary's advantage in both cost and SLO attainment results from Tributary's exploitation of spot instance characteristics (Sec. 5.3).

[4] At small scales, ExoSphere with low α had no resource diversity. With large α, it acquired too many resources, increasing its cost.

Proteus. We implemented Proteus's allocation strategy, described in Sec. 2.2, modified to acquire only spot resources (reducing cost with no significant change in SLO attainment). Fig. 5 compares Tributary and Proteus for the *ClarkNet* trace, for two different scaling policies. While Proteus achieves lower cost than Tributary, it experiences a large increase in "slow" requests. This increase is due to Proteus not diversifying its resource pool, instead only acquiring resources based on reducing average per-core cost. When told by the scaling policy to acquire additional resources, similarly to AutoScale buffers (Sec. 5.3), Proteus is unable to match Tributary's number of "slow" requests no matter how large the buffer (and, thus, how high the cost). This is once again due to the lack of diversity in the resources that Proteus acquires.

5.7 Prediction Model Evaluations

This section evaluates the accuracy of the preemption prediction models used by Tributary, which are described in Sec. 3.1. The recent Proteus system [16] used the historical median probability of preemption depending on the instance type, availability zone and the difference between the user bid price and the spot market price of the resource. Tributary improves prediction accuracy by using machine learning inference models trained with historical spot market data with engineered features. Fig. 6 shows the accuracy and F_1 scores for prediction models based on the historical median, a logistic regression classifier, a multilayer perceptron neural network (MLP NN) and a long short term memory recurrent neural network (LSTM RNN). These models were trained on spot market data from 06/06/16 – 01/22/17 and were evaluated on data from 01/23/17 – 03/20/17 for instance types *c4.large*, *c4.xlarge* and *c4.2xlarge* in *us-west-2*.

The output of the prediction models is whether the instance specified in a query will be preempted within the preemption window. Accuracy scores are calculated by the number of samples classified correctly divided by total number of samples. F_1 scores, which account for data skew, are a good accuracy measurement because the data set is skewed toward preemptions at lower *bid deltas* and non-preemptions at higher *bid deltas*. The LSTM RNN model provides the best accuracy and the best F_1 because it is able to capture the temporal nature of the AWS spot market. LSTM increases accuracy by 11% and the F_1 score by 27% compared to using the historical median. The MLP NN model performs worse than the historical median model for accuracy, but its F_1 score is higher because unlike the historical median model, the MLP model considers advanced features when predicting preemptions as described in Sec. 3.1. The increased accuracy of the LSTM RNN model translates to Tributary's effectiveness. When using the LSTM RNN model, Tributary runs at ≈37% less cost on the *ClarkNet* workload compared to Tributary using historical medians, because the historical median model overestimates the probability of preemption, causing Tributary to acquire more resources than necessary.

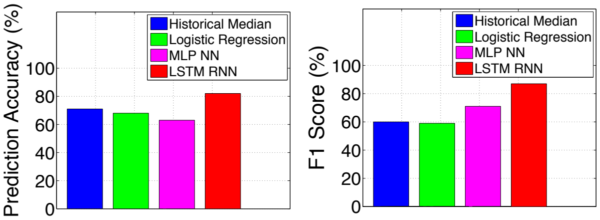


Figure 6: Accuracies and F_1 scores (accounts for data skew) for predicting preemption of AWS spot instances. The LSTM RNN outperforms prior techniques (blue bar) by 11% on the accuracy metric and 27% on the F_1 score metric.

6 Conclusion

Tributary exploits AWS spot instances to meet latency SLOs for elastic services at lower cost. By predicting preemption probabilities and acquiring diverse resource footprints, Tributary can aggressively use collections of cheap spot instances to reliably meet SLOs even in the face of bulk preemptions. Our experiments show cost savings of 81–86% relative to using non-preemptible on-demand instances and 47–62% relative to traditional high-risk use of spot instances.

Tributary exploits AWS properties, such as dynamic spot markets and preemption based thereon. We believe its approach would also work for other clouds offering preemptible resources, if they expose enough information to predict preemption probabilities, which AWS provides via the visible spot market prices. Currently, *Google Cloud Engine* [5] does not expose such a signal for its preemptible instances. For private clouds, exposing preemption logs could provide the historical view, but even better predictions can be enabled by exposing scheduler state.

Acknowledgements: We thank our USENIX ATC18 reviewers and our shepherd, Christopher Stewart, for valuable suggestions. We thank Henggang Cui for his feedback. We thank the members and companies of the PDL Consortium (Alibaba, Broadcom, Dell EMC, Facebook, Google, HPE, Hitachi, IBM, Intel, Micron, Microsoft, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma, Toshiba, Veritas and Western Digital) for their interest, insights, feedback, and support. This research is supported in part by Intel as part of the Intel STC for Visual Cloud Systems (ISTC-VCS), National Science Foundation under awards CNS-1042537, CCF-1533858, CNS-1042543 (PRObE [13]), and DARPA Grant FA87501220324. Lastly, one of the co-authors is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, and gifts from Alibaba, Amazon Web Services, Ant Financial, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft, Scotiabank, Splunk and VMware.

References

[1] Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot.

[2] AWS Autoscale. https://aws.amazon.com/autoscaling/.

[3] AWS EC2. http://aws.amazon.com/ec2/.

[4] AWS EC2 Spot Fleet. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet.html.

[5] Google Compute Engine. https://cloud.google.com/compute/.

[6] Spot Bid Advisor. https://aws.amazon.com/ec2/spot/bid-advisor/.

[7] Tensorflow serving. https://tensorflow.github.io/serving.

[8] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI 16).*

[9] Agmon Ben-Yehuda, O., Ben-Yehuda, M., Schuster, A., and Tsafrir, D. Deconstructing Amazon EC2 spot instance pricing. *ACM Transactions on Economics and Computation 1*, 3 (2013), 16.

[10] Danzig, P., Mogul, J., Paxson, V., and Schwartz, M. The internet traffic archive. *URL: http://ita.ee.lbl.gov/* (2000).

[11] Galante, G., and Bona, L. C. E. d. A survey on cloud computing elasticity. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing* (Washington, DC, USA, 2012), UCC '12, IEEE Computer Society, pp. 263–270.

[12] Gandhi, A., Harchol-Balter, M., Raghunathan, R., and Kozuch, M. A. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst. 30*, 4 (Nov. 2012), 14:1–14:26.

[13] Gibson, G., Grider, G., Jacobson, A., and Lloyd, W. Probe: A thousand-node experimental cluster for computer systems research. *USENIX 38*, 3 (2013).

[14] Gong, Z., Gu, X., and Wilkes, J. Press: Predictive elastic resource scaling for cloud systems. In *6th IEEE/IFIP International Conference on Network and Service Management (CNSM 2010)* (Niagara Falls, Canada, 2010).

[15] Group, W. N. R., et al. Wits: Waikato internet traffic storage. *URL: http://wand.net.nz/wits/index.php.*

[16] Harlap, A., Tumanov, A., Chung, A., Ganger, G. R., and Gibbons, P. B. Proteus: Agile ML elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 589–604.

[17] Hoxmeier, J. A., and DiCesare, C. System response time and user satisfaction: An experimental study of browser-based applications. *AMCIS 2000 Proceedings* (2000), 347.

[18] Kaliski, B. Pkcs# 5: Password-based cryptography specification version 2.0.

[19] Kohavi, R., and Longbotham, R. Online experiments: Lessons learned. *Computer 40*, 9 (2007).

[20] Liu, H., and Wee, S. *Web Server Farm in the Cloud: Performance Evaluation and Dynamic Architecture.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 369–380.

[21] Marathe, A., Harris, R., Lowenthal, D., De Supinski, B. R., Rountree, B., and Schulz, M. Exploiting redundancy for cost-effective, time-constrained execution of HPC applications on Amazon EC2. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing* (2014), ACM, pp. 279–290.

[22] Muller, K., and Vignaux, T. Simpy: Simulating systems in python. *ONLamp. com Python Devcenter* (2003).

[23] Sharma, P., Guo, T., He, X., Irwin, D., and Shenoy, P. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys 16)* (2016), ACM, p. 6.

[24] Sharma, P., Irwin, D., and Shenoy, P. Portfolio-driven resource management for transient cloud servers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems 1*, 1 (2017), 5.

[25] Sharma, P., Lee, S., Guo, T., Irwin, D., and Shenoy, P. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 16.

[26] Sharma, U., Shenoy, P., Sahu, S., and Shaikh, A. A cost-aware elasticity provisioning system for the cloud. In *2011 31st International Conference on Distributed Computing Systems* (June 2011), pp. 559–570.

[27] Shastri, S., and Irwin, D. Hotspot: Automated server hopping in cloud spot markets.

[28] Souders, S. Velocity and the bottom line. In *Velocity (Web Performance and Operations Conference)* (2009).

[29] Sundermeyer, M., Schlüter, R., and Ney, H. LSTM neural networks for language modeling. In *Interspeech* (2012), pp. 194–197.

[30] Tang, S., Yuan, J., and Li, X.-Y. Towards optimal bidding strategy for Amazon EC2 cloud spot instance. In *Proceedings of the 5th IEEE International Conference on Cloud Computing(CLOUD 12)* (2012), IEEE, pp. 91–98.

[31] Wang, W., Li, B., and Liang, B. To reserve or not to reserve: Optimal online multi-instance acquisition in iaas clouds. In *ICAC* (2013), pp. 13–22.

[32] Xu, Z., Stewart, C., Deng, N., and Wang, X. Blending on-demand and spot instances to lower costs for in-memory storage. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE* (2016), IEEE, pp. 1–9.

[33] Zheng, L., Joe-Wong, C., Tan, C. W., Chiang, M., and Wang, X. How to bid the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), ACM, pp. 71–84.

FastTrack: Foreground App-Aware I/O Management for Improving User Experience of Android Smartphones

Sangwook Shane Hahn, Sungjin Lee[†], Inhyuk Yee[*], Donguk Ryu[‡], and Jihong Kim
Seoul National University, [†]DGIST, [*]AIBrain Asia, [‡]Samsung Electronics

Abstract

The quality of user experience on a smartphone is directly affected by how fast a foreground app reacts to user inputs. Although existing Android smartphones properly differentiate a foreground app from background apps for most system activities, one major exception is the I/O service where I/O-priority inversions between a foreground app and background apps are commonly observed. In this paper, we investigate the I/O-priority inversion problem on Android smartphones. From our empirical study with real Android smartphones, we observed that the existing techniques for mitigating I/O-priority inversions are not applicable for smartphones where frequently inverted I/O priorities should be quickly corrected to avoid any user-perceived extra delay. We also identified that most noticeable I/O-priority inversions occur in the page cache and a flash storage device. Based on the analysis results, we propose a foreground app-aware I/O management scheme, called FastTrack, that accelerates foreground I/O requests by 1) *preempting* background I/O requests in the entire I/O stacks including the storage device and 2) *preventing* foreground app's data from being flushed from the page cache. Our experimental results using a prototype FastTrack implementation on four smartphones show that a foreground app can achieve the equivalent level of user-perceived responsiveness *regardless of the number of background apps*. Over the existing Android I/O implementation, FastTrack can reduce the average user response time by 94% when six I/O-intensive apps run as background apps.

1 Introduction

As a highly interaction-oriented device, a smartphone needs to react promptly *without a noticeable delay* to user inputs. In order to minimize a user-perceived delay, which directly affects the quality of user experience on the smartphone, Android smartphones properly differentiate a foreground (or FG) app from background (or BG) apps for most system activities. For example, when CPU cores are allocated, an FG app may be allowed to use one or more CPU cores exclusively while BG apps must share CPU cores with other apps [1, 2]. Such FG app-centric resource management becomes more important for modern Android smartphones because they run more apps at the same time thanks to aggressive multitasking support. As the number of concurrent BG apps increases, an FG app may encounter more interference from BG apps unless the FG app is managed with a higher priority over the BG apps.

Unlike FG app-aware CPU management which has been extensively investigated [3, 4, 5], I/O management on smartphones has not actively considered the quality of user experience issue in designing various I/O-related techniques. FG app-oblivious I/O management was not of a big concern for older smartphones where the number of BG apps is quite limited because of a small DRAM capacity (e.g., 512 MB) [6, 7, 8]. However, on modern high-end smartphones with a large number of CPU cores and a large DRAM capacity (e.g., 8 GB) [9, 10, 11] where the number of BG apps has significantly increased (e.g., from one in Nexus S [12] to more than 8 in Galaxy S8 [13]), FG I/O requests (or FG I/Os) are more likely to be interfered with BG I/O requests (or BG I/Os). Unless FG I/Os are treated with a higher priority over BG I/Os, FG I/Os may have to wait for the completion of a BG I/O. (That is, I/O-priority inversions occur.) In this paper, we comprehensively treat the I/O-priority inversion problem on Android smartphones including its impact on user experience, its main causes and an efficient solution.

Our work is mainly motivated by our empirical observation that I/O-priority inversions between the FG I/Os and the BG I/Os are quite common on Android smartphones. In particular, when an FG app needs a large number of I/Os (e.g., when the app starts), such I/O-priority inversions significantly degraded the response time of the FG app. For example, when five BG apps run at the same time, the app launch time of an FG app can increase by up to four times over when no BG app competes. This large increase in the app launch time of the FG app was rather surprising because the Android system is already designed to handle the FG app with a higher priority. In order to understand why the response time of a higher-priority FG app is affected by the number of BG apps, we have extensively analyzed the complete I/O path of the Android/Linux I/O stack layers on several smartphones using our custom I/O profiling tool, IOPro [14]. From our analysis study, we found that I/O-priority inversions in the page cache and the storage device were main causes of the increased response time of an FG app. We also observed that the current flush policy in the page cache, which does not distinguish whether a victim page (to be flushed) is from an FG app or a BG app, significantly impacted the FG app performance.

For frequent I/O-priority inversions on a smartphone, the existing techniques such as [15, 16, 17, 18, 19] may not be applicable because these techniques require a long latency (from the smartphone's viewpoint) to correct the inverted I/O priorities. For example, [19] depends on the priority inheritance protocol [20] to accelerate the completion of the current BG I/O before an FG I/O is started. In our experiment, the FG I/O waited up to 117 ms for the completion of the current BG I/O under the priority inheritance protocol. Obviously, this is too long for a smartphone where a delay of more than a few milliseconds is unacceptable [19]. Furthermore, the efficiency of these existing techniques, which were not specifically developed for smartphones as a target system, is limited in several aspects. For example, they do not fully exploit an important hint such as the type of apps (e.g., foreground or background) and do not take a holistic approach in optimizing the entire I/O stack layers *including the storage device*. Therefore, a different approach is necessary for resolving the I/O-priority inversion problem on smartphones. A solution should meet the fast response time requirement and should better exploit the smartphone-specific hints in a holistic I/O-path-aware fashion.

In this paper, we propose a new I/O management scheme for Android smartphones, called FastTrack (or FastT in short), which efficiently meets the above requirements on resolving I/O-priority inversions on Android smartphones. The key difference of FastTrack over the existing techniques is that FastTrack takes a more direct approach in fixing the I/O priority inversion problem by *preempting* the current background activity throughout the entire I/O stack layers. By stopping the current background activity *immediately*, FastTrack can quickly service the I/O request from an FG app. FastTrack also modifies the flush policy of the page cache to be *FG app-aware*. For example, when a victim page is selected for the next flush, FastTrack first considers pages that belong to BG apps as victim candidates.

In order to evaluate the effectiveness of the proposed scheme, we have implemented FastTrack on various Android smartphones, including Nexus 5 [21], Nexus 6 [22], Galaxy S6 [23] and Pixel [24]. Our experimental results show that FastTrack can provide the equivalent level of responsiveness to FG apps regardless of the number of BG apps. For important I/O-intensive app use cases (such as the app launch time, app switch time and app loading time)[1], compared over when no BG app runs, FastTrack can limit an increase in the average response time of an FG app within 27% even when six I/O-intensive BG apps run together. On the other hand, in the default Android implementation, the average response time can increase up to by 2,319%. Because of foreground app-centric I/O management, FastTrack is very effective in decreasing the average response time of an FG app as well. For example, when six BG apps run together, FastTrack can reduce the response time of an FG app by 94% over the default Android implementation.

The remainder of this paper is organized as follows. In Section 2, we report the key results of our empirical study on the impact of BG I/Os on user experience. Section 3 describes the root causes of the I/O-priority inversion problem on smartphones and summarizes the main requirements that must be satisfied by a solution. A detailed description of FastTrack is given in Section 4. Experimental results follow in Section 5, and related work is summarized in Section 6. Finally, Section 7 concludes with future work.

2 Impact of BG I/Os on User Experience

In this section, we empirically analyze how much FG app performance is affected by BG I/Os. We also investigate how often BG I/Os interfere with an FG app. Our empirical study is carried out under various real-life smartphone usage scenarios with 10 different smartphones.

2.1 Evaluation Study Setup

For our study, we collected 10 Android smartphones[2] (with a proper instrumentation function) from different users who are all *heavy* users (almost always carrying their smartphones with them). To avoid possible bias, we have selected the smartphones from seven different manufacturers. Each smartphone is equipped with 4 or more cores and 3 GB or larger DRAM memory which is large enough to actively support multitasking. All of the smartphones also have the latest version of Android (version 7.x) which supports enhanced multitasking features (such as a split screen).

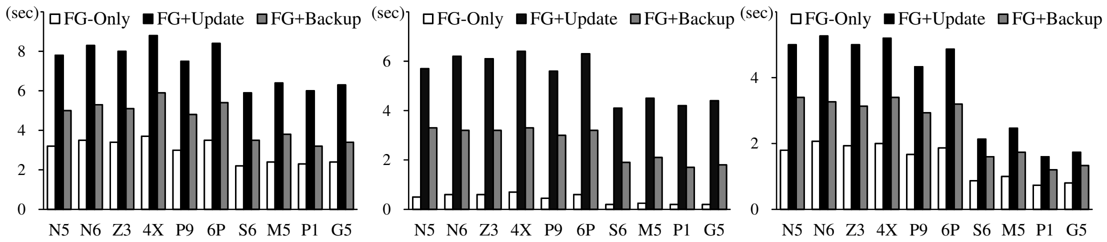
Like other active smartphone users, our study participants used Chrome, Messenger, Camera, Gallery, and Game as their main FG apps. As popular BG apps, cloud backup apps such as Dropbox [25] and OneDrive [26] were popular among the study participants. Furthermore, all the participants enabled an option for an automatic app update. The "background process limit" option was set to "standard limit", which is a default setting.

2.2 Response Time Analysis

In order to understand an impact of BG I/Os on user experience, we conducted a series of experiments on common smartphone usage cases when typical BG apps run. We have measured the user-perceived response times

[1] Since a user must wait for these use cases to complete, their response times directly affect the smartphone user experience quality.

[2] 10 phones include Nexus 5 (N5), 6 (N6), 6P (6P), Z3 (Z3), Redmi 4X (4X), P9 (P9), Galaxy S6 (S6), Mi 5 (M5), Pixel (P1), and G5 (G5).



(a) Gallery app launch time.

(b) Camera app switch time.

(c) Game app loading time.

Fig. 1: Impact of background I/Os on user experience.

of three usage cases where prompt reaction to user inputs are important – when (i) a new app is launched by a user, (ii) one app is switched to another, and (iii) a launched app is loading required contents. For BG apps, we have selected two BG workloads: `Update` and `Backup`. `Update` downloads and installs multiple apps from the Android market (e.g., as in an auto app update), and `Backup` uploads a large number of files to cloud storage services (e.g., as in Dropbox). These workloads were selected because they are known to generate substantial BG I/Os in a periodic fashion[3] Our background scenarios are automatically invoked in background when a smartphone is connected to a fast network (e.g., Wi-Fi).

Scenario A – Launching `Gallery` **App:** Launching an app requires to load a relatively large number of files, including executables, libraries, and files. While an app is being launched, a user has to wait until all the required files are loaded from a storage device. In this paper, an app launch refers to a cold start, where an app is launched without any preloaded data. It should be noted that, as the quality and complexity of mobile applications improve, the amount of data to be loaded while launching apps increases as well. In case of `Gallery` app [30], for example, 25 image files, on average, must be preloaded to complete the app's initial display.

Fig. 1(a) depicts the launch time of `Gallery` on the 10 smartphones. Here, the launch time is defined to be the time interval from when an app icon is touched by a user to when all the components are displayed on the screen. Even though there are differences depending on the hardware performance, noticeable launch time degradations are observed in all the smartphones when BG I/Os are issued simultaneously. In N6 with an eMMC storage, the launch times under two BG workloads, `FG+Update` and `FG+Backup`, increase by 2.4 times and 1.6 times, respectively, over a standalone launch (`FG-only`). Similar trends are also observed even in smartphones with faster mobile storage systems. In S6 with an UFS storage that provides higher throughput, the launch times of `FG+Update` and `FG+Backup` increase by 2.6 times and 1.7 times, respectively.

Scenario B – Switching `Camera` **App:** Switching from one app to another becomes a common feature in smartphones supporting multitasking. Before moving to a new app, a current app should be properly suspended. In the Android platform, the app switch involves the flushing of dirty pages in the page cache to persistent storage, so as to create as many free pages as possible for a new app. A user must wait for an old app to complete flushing its dirty pages before a new app is activated. Therefore, a user may experience a long unpleasant delay between app switches if BG I/Os interfere with the flushing process in the page cache.

We examine the switch time of a `Camera` app [31] when it switches to a home screen app. While the `Camera` app is recording a video for 10 minutes, we measure the time interval from when the home button is pressed to when the home screen is displayed for the next user interaction. Fig. 1(b) illustrates the switch time of `Camera` in S6 – it is less than 1 second when no BG I/Os are being issued, but it increases by 19.5 times under heavy BG I/Os. In N6, the switch time also increases by 11 times compared to when there are no BG I/Os.

Scenario C – Loading `Game` **App:** After app launching, some apps require additional file loading work before a user interacts with launched apps. One representative example is a `Game` app [32] that has to preload game contents (e.g., stage maps and rendered images) depending on a user's input after it completed the app launching process. This loading process inevitably results in response time delays from the perspective of end users.

In order to understand how much BG I/Os affect the app loading time, we measure the time interval from when the 'story mode' button on a `Game` app [32] is touched by a user to when its loading process is finished. As expected, we observed that the app loading time increases with BG I/Os in all the smartphones. We also confirm that the app loading times tend to be longer in smartphones with less memory over ones with larger memory. For example, on N5 with 2 GB DRAM, the loading time increases by 2.7 times under `Update`. The loading time, however, increases about 2 times only on P1 with 4 GB DRAM.

[3] `Update` is based on an observation that popular mobile apps (such as `Twitter`) are typically updated once every week and the average size of downloaded packages for an app update is about 110 MB [27, 28], and `Backup` is based on a report that a typical smartphone user uploads more than 200 MB of files per day to the cloud storage [29].

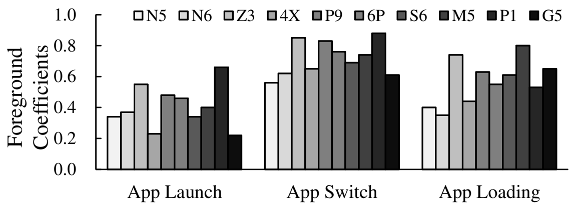


Fig. 2: FG-BG interference analysis results.

2.3 FG-BG Interference Analysis

Although we confirmed that BG I/Os can significantly degrade the quality of user experience of an FG app when they conflict with FG I/Os, if BG I/Os were unlikely to overlap with FG I/Os in practice, our response time analysis in Section 2.2 becomes less meaningful. For example, if most BG I/Os occurred while a smartphone was not actively used by a user, their actual impact on user experience would be negligible, thus making our work useless. In order to evaluate if such conflicts are really happening in real-world settings, we built a simple I/O utility[4] which can tell how much BG I/Os were issued while processing a given FG I/O *req*. If an FG I/O τ was started at t_{start} and completed at t_{finish} , our utility computes the ratio r_τ of the total amount of I/Os from τ to the total amount of I/Os in the interval $[t_{start}, t_{finish}]$. This ratio, we call the foreground coefficient C_{fg}, indicates the proportion of FG I/Os over the total I/Os in $[t_{start}, t_{finish}]$. If C_{fg} is high, it indicates that there is less interference from BG I/Os. For example, if C_{fg} is close to 1, few BG I/Os interfere with the FG I/O.

Fig. 2 shows how much BG I/Os interfere with FG apps on 10 smartphones. For each smartphone, we have collected a month's history of system call usage and computed average foreground coefficients for three use cases (explained in Section 2.2). Fig. 2 shows that a significant portion of BG I/Os can interfere with user's interaction with FG apps. For example, in the app launch scenario, FG I/Os account for only 42% of the total I/Os, which can conflict with 58% of the total I/Os that are requested from BG apps. Similarly, in the app switch scenario and the app loading scenario, FG I/Os are responsible for 77% and 53% of the total I/O requests, respectively. Although the BG I/O portion was reduced over the app launch scenario, the BG I/O portion is still large enough to affect the user experience of an FG app in a significant fashion.

3 Root Causes of User-Perceived Delay

In this section, we analyze the I/O stack of the Android platform to find root causes that are responsible for rapidly increasing user-perceived response time along with an increasing number of I/O-intensive BG apps. We first review the overall architecture of the Android I/O stack, giving a brief explanation of how apps access files in storage media. Then, we explain three major bottlenecks we found through the analysis.

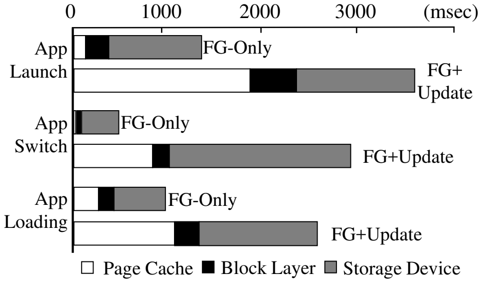


Fig. 3: A breakdown of foreground I/O execution time.

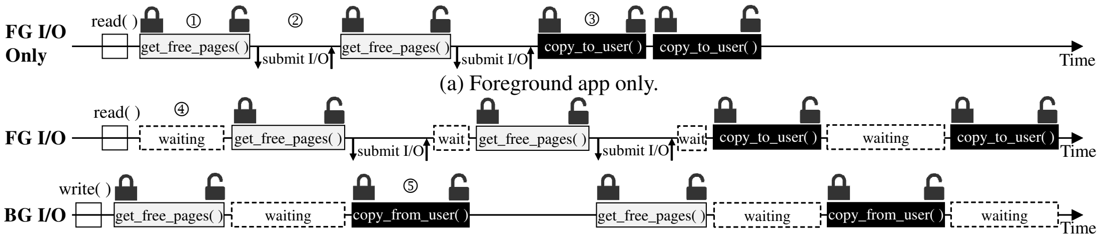
3.1 Overview of Android I/O Stack

As in typical UNIX-like OSes, Android file I/Os (i.e., reads and writes on files) created by an app are delivered to the kernel through system-call interfaces. The Linux kernel checks if corresponding file data is already cached in the page cache. If not, free pages available in the page cache are allocated to individual file I/Os. If the file I/O is for writes, user data is copied to the allocated pages in the kernel. Before sending I/O commands to an underlying block device, each file I/O is converted into a set of block I/Os with designated logical block addresses (LBAs). Block I/Os are then transferred to the block I/O layer and are put into proper I/O scheduling queues, sync or async queues, according to their types. I/O scheduling algorithms (e.g., CFQ [34]) move ready-to-submit block I/Os to a dispatch queue, which will be sent to the storage device via the eMMC [35] or UFS [36] interface. If the file I/O is for reads, data read from the storage device is stored in the allocated pages, and the data is finally copied to the user-space buffer.

In order to analyze the root causes of performance degradation by BG I/Os, we have measured the execution time of FG I/Os using IOPro. IOPro is capable of measuring the detailed elapsed times of I/O requests across all the Android I/O stacks, including a page cache, a block I/O layer, and a storage device. Fig. 3 shows a breakdown of the I/O execution time observed in the three usage scenarios used in Section 2. Because of the space limit, only the results from S6 are displayed in Fig. 3, but other smartphones also exhibit similar performance trends. When there are no BG I/Os (denoted by `FG-Only` in Fig. 3), the storage device is a major bottleneck. This is a reasonable result because the storage device is considered the slowest component in the I/O stack hierarchy.

With BG apps running heavily (`FG+Update` in Fig. 3), we observe that the execution times increased consider-

[4] Our monitoring tool is based on strace which is a popular profiling utility for analyzing system calls [33]. Strace provides PIDs of processes that generate I/Os, along with detailed information of relevant I/O system calls. Using the collected information, we can distinguish BG I/Os from FG IOs with their respective I/O traffic amounts.



(a) Foreground app only.

(b) Both foreground app and background app.

Fig. 4: Impact of lock contention on the I/O latency of the foreground app.

ably across all the layers. In particular, the times spent by the page cache layer have increased by 12 times, on average. For example, in Fig. 3, the portions of the page cache in `FG-Only` were negligible, but these rapidly increased in `FG+Update` to 32%-51% of the total execution times. While the relative portion of the time spent by the storage device has decreased, the total execution time spent by the storage device has significantly increased. For example, in the case of the app switch scenario, the execution time on the storage device has increased by 5 times. One surprising result in our study was that the impact of the block I/O layer on performance was rather negligible compared with the other two layers.

In the following three subsections, we investigate what happens inside the kernel I/O stacks when BG I/Os are heavily issued. Particularly, we focus on analyzing internal I/O activities at the page cache and the storage device layers because they are the main contributors to the increase of the execution times.

3.2 Root Cause 1: Page Allocation

From our performance bottleneck study, we found that *lock contentions in the page cache layer* are responsible for many I/O-priority inversions we have observed. As the first and major root cause of a performance degradation of an FG app under BG apps, we explain the impact of the page allocation module on user experience. When a new I/O request arrives at the kernel, free pages should be assigned first to the I/O request. When new free pages are necessary for serving the incoming I/O request, a free-page allocation module first acquires a *global lock*, *page lock*, for the exclusive access of the page cache during the page reclamation process [37] which is *non-preemptive* [38]. Acquiring free pages is mostly done quickly. However, if there are not sufficient free pages available, it takes a rather long time (e.g., more than 200 ms [19]) to create free pages by evicting dirty pages. Evicting dirty pages require extra writes to the storage device. If FG I/Os are blocked by BG apps that need the free page reclamation process, an FG app has to wait for BG I/Os to finish, thus causing an I/O-priority inversion between the FG app and the BG apps.

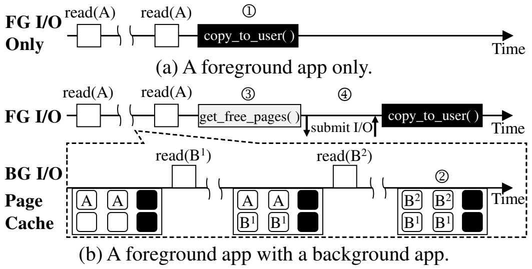
Fig. 4(a) illustrates an example where an FG app *F* reads a photo file of 256 KB size from storage media by calling a `read()` system call. We compare two different cases: 1) when *F* runs alone without any BG apps and 2) when *F* runs together with a BG app *B* that writes a large file to the storage device. Without BG apps, the FG app can quickly get free pages from the page cache (by calling `alloc_pages()` ①). Since the maximum allocation unit of free pages is limited to 128 KB [39], the kernel calls `alloc_pages()` twice, each of which gets 128 KB free pages. After calling each `alloc_pages()`, the kernel sends a read I/O command to the storage device (②), which transfers file contents from the storage to the allocated pages. Finally, data kept in the kernel pages are copied to a user-space buffer in the unit of 128 KB (by calling `copy_to_user()` ③).

Suppose that the BG app calls the `write()` system call to write data just before `read()` is invoked by the FG app. The *page lock* is grabbed by the BG app first, so the FG app has to wait until it releases the *lock* (④). This could be quite long if dirty page evictions are involved while assigning free pages to the BG app. After the *page lock* is released by the BG app, the FG app is able to acquire the lock, allocates free pages for reads, and then releases the lock. Then, it issues a read I/O command to the device. Copying data from the user space to the kernel space (`copy_from_user()`) also requires holding the same *global lock* of the page cache (⑤). As depicted in 4(b), if the BG app has already acquired the *global lock*, the FG app has to wait again for the lock to be released, which increases additional user-perceived delays.

Some might argue that the eviction of dirty pages in the middle of calling `alloc_pages()` would rarely occur. In our observation, however, when write-dominant BG apps run (e.g., `Update`), many dirty pages are created in the page cache and available free pages quickly run out. If an FG app requests I/Os in such situations, frequent evictions of dirty pages are inevitable.

3.3 Root Cause 2: Page Replacement

Our second root cause comes from a somewhat surprising source. As discussed in Section 3.2, the performance degradation of an FG app from the lock contention mostly occurs when many dirty pages are created in the page cache. When BG apps are read-dominant, such performance degradation is difficult to occur because few dirty pages may exist in the page cache. For example, in ④ of Fig. 4(a), if reclaimed pages were



(a) A foreground app only.

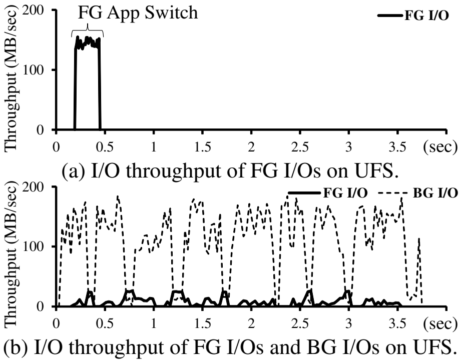
(b) A foreground app with a background app.

Fig. 5: Impact of page replacement.

clean, no writes to the storage device would be needed. Unlike our reasoning about read-dominant BG apps, our experiments revealed an interesting result that *even read-dominant BG apps can often interfere with an FG app*.

Although it was not straightforward to understand why such unexpected performance degradation occurs, we identified the page replacement policy in the page cache as the main cause. The existing Linux page replacement policy in the page cache works in an FG app-oblivious fashion. That is, the Linux kernel prefers choosing a clean page as a victim because of its cheap replacement cost regardless of whether the owner of the victim page is an FG app or a BG app. Suppose that BG apps want to read a large amount of data from the storage and they need to get more free pages by evicting existing ones from the page cache. In this situation, the Linux kernel often selects clean pages of an FG app even though those clean pages are soon to be accessed. Although choosing a clean page as a victim page is reasonable from minimizing the eviction cost, it is a bad decision for the FG app because a large page cache miss penalty can significantly increase the FG app response time.

Fig. 5 shows a concrete example of how a read-dominant BG app negatively affects an FG app. Here, the FG app *F* is assumed to read a file A twice by calling `read()`. Again, we compare two different cases: 1) when *F* solely runs and 2) when *F* runs together with a BG app which read a large file B from the storage device. Without BG apps, the FG app can quickly finish the second `read()` by reading the file A from the page cache (①). However, when the FG and BG apps run simultaneously, some pages of the file A may be evicted from the page cache (②) to create a room for the large file B. After the completion of BG reads, when the FG app tries to read the file A again, free pages should be allocated (③) and the previously-evicted pages should be read from the storage device again (④). Even worse, from our investigations on real-life app usage scenarios, we observed that many FG apps exhibit high temporal locality, repeatedly referencing the same files. For such an FG app, the existing page cache replacement policy can significantly degrade the user experience by evicting performance-critical hot pages of the FG app.



(a) I/O throughput of FG I/Os on UFS.

(b) I/O throughput of FG I/Os and BG I/Os on UFS.

Fig. 6: I/O priority inversions in flash storage device.

3.4 Root Cause 3: Device I/O Scheduling

After our bottleneck study on the page cache layer, we investigated the block layer as a next candidate for the I/O priority inversion problem. We analyzed how the block layer processes I/O requests from when the I/O requests are put into the I/O scheduler queue to when an interrupt handler receives signals notifying the completion of the requests in the storage device. Our investigation revealed that the I/O priority inversion problem occurred *in the storage device* rather than in the block layer.

Once the storage device gets I/Os from a block device driver, it processes them according to its own I/O scheduling algorithm. The storage device generally gives a higher priority to reads than writes because reads have a higher impact on user-perceived response time. For the same type of I/O requests, the storage device process them in an FIFO manner with no preference. Although this generic scheduling policy works reasonably well for equal-priority I/O requests, it causes I/O-priority inversions very frequently because the scheduling policy inside the storage device is not aware of the priority of an I/O request. For example, if FG writes and BG reads are sent to the storage device, the FG writes would be delayed until all the BG reads complete.

Fig. 6 illustrates the negative impact of a priority-unaware I/O scheduler inside a storage system on the throughput of FG I/Os. It plots the throughputs of FG I/Os and BG I/Os in the app switch scenario, where an FG app writes a large number of files, while huge files are being read in background. Note that the I/O throughputs were measured at the block device driver in order to device-level performance. Unlike the `FG-Only` case shown in Fig. 6(a), a significant degradation of the FG I/O throughput is observed in Fig. 6(b) when FG writes and BG reads are mixed inside the storage system. The app switch scenario, which was completed in 0.45 seconds without BG I/Os, took 3.55 seconds to finish. Our additional experiments showed that the I/O-priority inversion problem within the storage device occurs very frequently whenever FG writes are mixed with BG reads and its impact on an FG app is very serious.

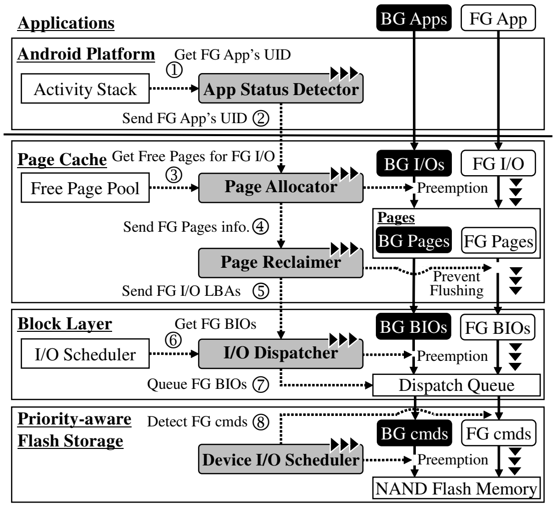


Fig. 7: An overall architecture of FastTrack.

4 Design and Implementation of FastTrack

As explained in the previous section, high-priority FG I/Os are unintentionally delayed by low-priority BG I/Os for various reasons across the entire I/O stack. One of the most commonly used solutions to resolve the I/O-priority inversion problem is to use the priority inheritance protocol that raises a priority of BG I/Os. The priority inheritance, however, is not effective in our cases – it still requires an FG app to wait for BG I/Os to finish, creating long delays to latency-sensitive smartphone users.

An ideal solution to resolve the problem is to create a *vertically-integrated fast I/O path* which is dedicated to serving FG I/Os across the entire I/O stack, including a page cache, a block layer, and a storage device. In other words, if it is possible to quickly preempt BG I/Os upon the arrival of FG I/Os and to deliver them directly to the storage device with minimal interference by I/O stack layers, it would be possible to provide the equivalent level of user-perceived responsiveness as when there is no BG I/O. Key technical challenges here are (1) how to identify FG I/Os from BG I/Os inside the kernel, (2) how to preempt BG I/Os immediately, and (3) how to prevent potential side effects that could occur when creating such a new I/O path.

Keeping these technical challenges in mind, we design the app status-aware I/O management, FastTrack, with five modules as illustrated in Fig. 7. The app status detector obtains the information of the current FG app by monitoring the activity stacks of the Android platform (①) and forwards it to the page allocator (②). Using this, the page allocator is able to identify I/O requests from the FG app, suspending the currently executing BG I/O jobs. The page allocator then grabs a global lock of the page cache, preferentially assigning free pages to FG I/Os, regardless of their arrival time (③). Until the page allocator releases the lock, BG I/Os are postponed.

If there are not enough free pages to handle I/O requests, the page reclaimer evicts kernel pages that belong to BG I/Os as victims, preventing FG pages from being flushed from the page cache (④). After acquiring all the free pages required, the page reclaimer builds up block I/Os for FG I/Os (FG BIOs) with designated LBAs, putting them into I/O scheduler's queue in the block layer (⑤). Upon the arrival of FG BIOs, the I/O dispatcher suspends servicing BG BIOs by limiting I/O queueing and then immediately delivers FG BIOs to the dispatch queue (⑥ and ⑦). When FG BIOs are converted to FG commands (FG cmds) for the storage device, the I/O dispatcher tags an FG I/O flag so that the device I/O scheduler suspends the BG I/O execution (⑧), and FG cmds can be processed immediately in the storage device.

4.1 App Status Detector

In order to identify an FG app among all the apps available in the system, the app status detector inquires of the Android activity manager holding all of the activities initiated by a user. Whenever a user inputs a command to a phone by touching a screen or an icon, the Android platform creates a new activity, which is a sort of job corresponding to user's command, and puts it into an activity stack in the Android activity manager. Since the top activity on the stack points to the current interactive app with a user (i.e., an FG app), the FG app information in the system can be easily retrieved.

All of the Android apps have its own unique ID number, called UID, which is assigned when an app was installed in the system. An UID number is different from Linux's process ID (PID). Thus, our next step is to find a list of the Linux processes connected to the FG app. A list in question can be obtained by examining all the processes in Linux's process tree. However, such an exhaustive search on the process tree takes a relatively long time. Therefore, the app status detector maintains an UID-indexed table that is updated whenever a new process or thread is created or terminated. Then, using UID as a key, the app status detector quickly retrieves a list of FG app's processes.

Whenever the top activity changes, the app status detector sends an UID of the new FG app, along with PIDs and TIDs of related Linux processes, to the Linux kernel via the `sysfs` interface. By doing this, app status detector can keep track of the currently executing FG app.

4.2 Page Allocator

The page allocator is designed to preferentially allocate kernel pages to I/O requests from an FG app by suspending outstanding BG I/Os. Fig. 8 shows how the page allocator works using the same example in Fig. 4, where the FG app generates read requests to the kernel just after the BG app issued write requests. The page allocator sees if the I/O request is from the FG app or not by

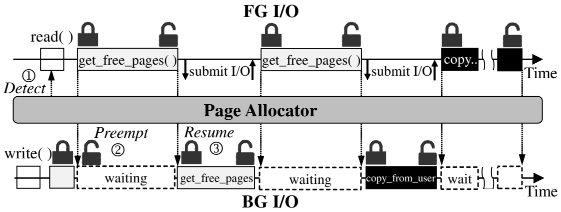


Fig. 8: Preemption of background I/Os.

comparing its UID, PIDs, and TIDs with the ones that it previously got from the app status detector (①). If it is from the FG app, the page allocator forces BG I/Os to release a global lock of the page cache just after getting a page currently being requested (②). After allocating desired pages to the FG I/Os, the page allocator resumes the preempted BG I/Os (③). At the same time, the kernel issues FG BIOs to fill up the allocated pages with data read from storage media. In a similar way, the page allocator suspends and resumes data copy operations of BG I/Os between the user and kernel space.

In order to support the prompt preemption and resumption of BG I/Os, we modified the major kernel functions relevant to the page cache, including `alloc_pages()`, `do_generic_file_read()`, and `generic_perform_write()`. These functions are divided into several execution segments. At the end of each segment, the page allocator checks if there are waiting FG I/Os. If so, the page allocator promptly suspends BG I/Os, unlocks the page-cache lock, and yields the CPU for the FG I/Os. After serving FG I/Os, the suspended BG I/Os restart at the point where they were suspended.

While conceptually simple, the implementation of the preemptive page cache raises two technical issues. Firstly, giving the highest favor to FG I/Os does not guarantee the improved response time all the time, and, in the worst case, it may result in serious response time degradation or even application deadlocks. Imagine an application that downloads files from the network and performs certain operations on the download files. The application model of Android requires an app to offload such a typical task to a built-in process that runs as a background service. In case of a file download, a network service process performs downloading files in background on behalf of a user app. If I/O requests from the network service process are preempted for FG I/Os, the execution of the FG app that initiates the file download would be delayed for a long time. Fortunately, the Android system does not allow such dependency between conventional user apps (e.g., game and camera apps) [40]. To avoid the self-harming preemption mentioned above, therefore, it is only necessary to prevent the preemption for BG I/Os from service processes. To do this, the page allocator checks if BG I/Os are issued by services or not and excludes them from the preemption if they are from service processes. This I/O filtering can be done by checking UID because the Android system assigns predefined UIDs to service processes, giving UIDs ranging from 10,000 to 19,999 to user apps [41].

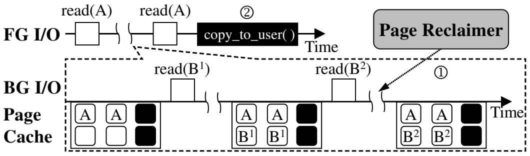


Fig. 9: Prevention of foreground page evictions.

Secondly, performing the preemption at the page cache level is not always possible. Some pieces of the kernel code must run in a special context, called an *atomic context* [42], which does not allow a CPU to go into sleep. Representative examples are interrupt handlers and critical section codes wrapped by spinlocks. The page allocator modifies the page cache functions that are also invoked by other parts of the kernel for various purposes. Thus, the page allocator should disable the preemption if it is called by a caller running in the atomic context. It is straightforward to know whether the page allocation is requested inside the atomic context. In the Linux kernel, a caller function should let the memory allocator know which type of contexts it runs now as an argument (e.g., `GFP_ATOMIC`). The page allocator refers this information and prevents the preemption if the allocation is requested inside the atomic context.

4.3 Page Reclaimer

In addition to preempting BG I/Os to accelerate page allocation for FG I/Os, the page reclaimer improves the performance of FG apps by preventing the eviction of kernel pages belonging to the FG apps.

Fig. 9 illustrates how the page reclaimer operates using the same example in Fig. 5, where the FG app attempts to read the file A twice, while the BG app is heavily reading the large file B. In Fig. 5, the second read to the file A is not hit by the page cache since it was chosen as a victim and was evicted from the page cache (② in Fig. 5). As explained earlier, this is due to the kernel's page replacement that evicts clean pages first, regardless of the status of an app. The page reclaimer prevents such a problem by adopting new replacement priorities for victim selection: *BG clean pages* (highest) $>$ *BG dirty pages* (high) $>$ *FG clean pages* (low) $>$ *FG dirty pages* (lowest). With the new policy, clean pages belonging to BG apps are preferentially evicted when there is insufficient memory. In Fig. 9, the pages labeled by B^1 are evicted for B^1, even though the pages for the file A were least recently referenced (① in Fig. 9).

Keeping FG pages in the page cache wouldn't be effective if an FG app has low temporal locality. In the worst case, it would degrade the performance of BG apps without any performance improvement on an FG app. According to the mobile app workload study [43], however, the majority of the apps have high degrees of data locality. Thus, the negative effects of the page reclaimer are expected to be minimal in smartphone usages.

4.4 I/O Dispatcher

The primary goal of FastTrack is to create a fast I/O path for FG I/Os in the entire kernel layer. To this end, it is also required to enhance the block I/O layer, together with the page cache layer. Once block I/Os are delivered to the block layer from the page cache, they are put into a sync queue or an async queue in the I/O scheduler according to their types. To accelerate FG BIOs, the I/O dispatcher looks for FG BIOs in both queues and moves them to the dispatch queue immediately.

Depending on the type of a queue, the I/O dispatcher has to take different strategies to find FG BIOs. FG BIOs can be easily found in the sync queue using the FG app's PID number delivered by the app status detector. In the case of async I/Os, however, the PID number of all async I/Os is the same as the PID of the kworker kthread which delivers async BIOs to the block layer on behalf of FG processes. Since the PID number is useless to find async FG BIOs, the I/O dispatcher uses LBAs as keys to fetch FG BIOs from the async queue.

Finally, whenever a new BIO enters the sync/async queues, the I/O dispatcher prevalidates whether it is FG BIO, then directly sends FG BIO to the dispatch queue regardless of its priority in sync/async queues.

4.5 Device I/O Scheduler

In order for FastTrack to achieve its maximum benefit, a storage device, which is at the lowest layer in the I/O stack, needs to be enhanced as well. According to [44, 45, 46], modern flash storage maintains its own internal queue, but is unaware of the status of applications issuing I/Os. To make a storage device FG I/O-aware, we modify an SCSI command set so that it carries an additional flag in a reserved opcode [47] that specifies whether I/O requests are issued by FG apps or not. This flag is used as a hint for a device-level I/O scheduler to decide the execution order of I/O requests staying in the internal I/O queue. In our current design, we assign the highest priority to FG reads, followed by FG writes and BG reads. BG writes are assigned to the lowest priority.

5 Experimental Results

In order to quantitatively evaluate the effect of FastT, we implemented the FastT modules in the Android 7.1.1 and the Linux kernel 3.10.61. Four smartphones, Nexus 5 (N5), Nexus 6 (N6), Galaxy S6 (S6) and Pixel (P1) were used for our evaluation. N5 and N6 use eMMC-based storage devices while S6 and P1 employ UFS-based storage devices. (Note that UFS supports 3 times higher throughput over eMMC.) All the smartphones were connected to the Internet through a 5-GHz Wi-Fi.

We have chosen two background usage scenarios: `Update` for a write-dominant workload and `Backup` for a read-dominant workload. The `Update` scenario updated Hearthstone game [48] downloaded from Play Store, whose size was about 1.5 GB. The `Backup` scenario uploaded 1 GB of data files to cloud storage.

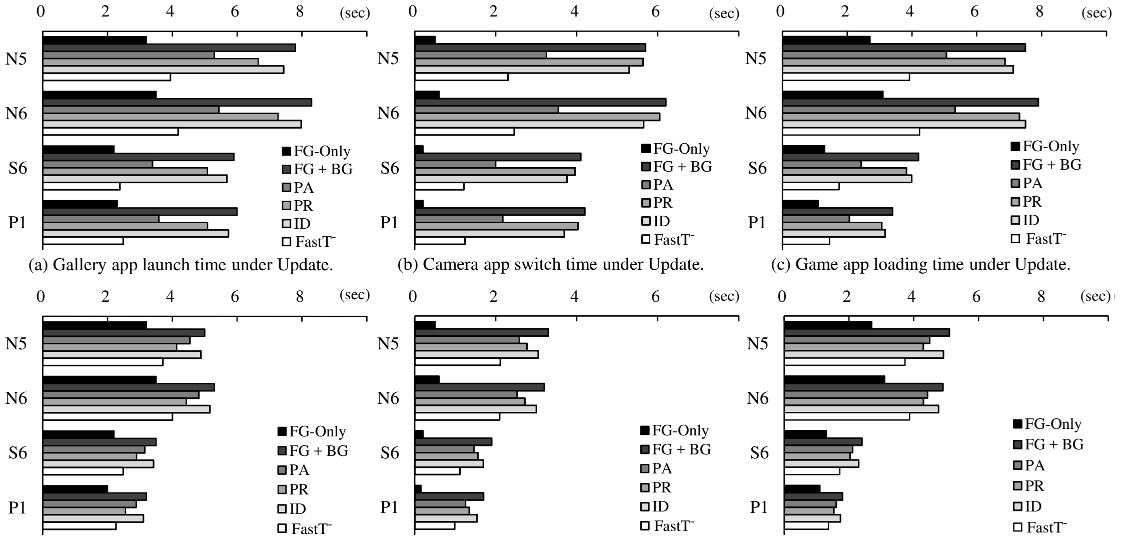
While running BG apps, we executed three FG apps, `Gallery` (app launch), `Camera` (app switch), and `Game` (app loading) discussed in Section 2.2. `Gallery` was a read-dominant workload, `Camera` was a write-dominant workload, and `Game` was a mixed workload. In order to accurately measure performance, all other apps were terminated before the experiment.

5.1 Performance Analysis on Smartphones

As the response time lower limit of an FG app, we first measured user-perceived response time of the FG app when only the FG app ran without any BG apps. To understand the impact of a BG app on performance, we also measured performance when both FG and BG apps ran simultaneously on the unmodified kernel. The above two cases are denoted by FG-only and FG+BG, respectively. We compared the performances of FG-only and FG+BG with four different versions of FastT: PA, PR, ID and FastT^{-}. PA, PR, ID represents FastT with a single main component only, that is, the page allocator, the page reclaimer, and the I/O dispatcher only, respectively. FastT^{-} employs all of the main components but it uses the existing storage device I/O scheduler[5]. In all the FastT versions we tested, the app status detector was enabled by default.

Fig. 10 shows that, for six different combinations of FG and BG apps, FastT^{-} reduced the user-perceived response times by 74% over FG+BG, on average. PA exhibited significant performance improvements when BG apps were write-dominant (i.e., `Update`). `Update` required a copy of data from the user space to the kernel, which involved the allocation of free pages in the page cache. PA not only made this acquisition process preemptible, but also gave a higher priority to an FG app so that it got free pages prior to BG apps. By doing this, PA was able to prevent FG I/Os from being blocked by BG writes. Unlike PA, PR mostly contributed to reducing user-perceived response time when the read-dominant BG app (i.e., `Backup`) ran. In our observation, `Backup` required many free pages to load files from the storage before sending them to cloud storage. To create free pages, it often selected clean pages belonging to FG apps as victims, which resulted in the eviction of hot data from the page cache. PR prevented those clean pages from being evicted from the page cache, thereby reducing the

[5] Unfortunately, we cannot access the firmware inside the storage device. For a complete FastT implementation, we use an emulated storage as discussed in Section 5.2.



(a) Gallery app launch time under Update. (b) Camera app switch time under Update. (c) Game app loading time under Update.

(d) Gallery app launch time under Backup. (e) Camera app switch time under Backup. (f) Game app loading time under Backup.

Fig. 10: Response time analysis on smartphones.

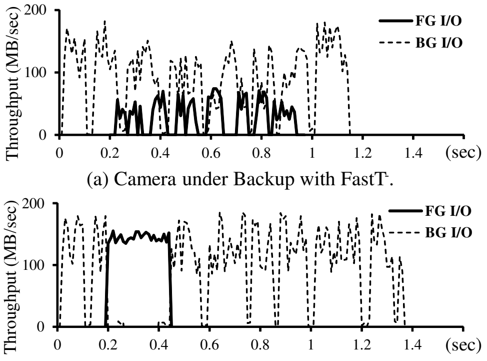
number of reads from the storage which were not necessary when only FG apps ran. ID improved the response times by 11% on average, but its impact on performance was negligible compared with PA and PR. This result confirmed our hypothesis that rescheduling I/O requests at the scheduler level was less efficient than doing it at a higher level – a page cache. As expected, by integrating the three techniques, FastT$^-$ exhibited the best performance among all the versions evaluated.

Fig. 10 also shows that FastT$^-$ works more efficiently atop a faster storage device like UFS (used in S6 and P1) than a slower one like eMMC (used in N5 and N6). In our observation, the absolute numbers of I/O latencies reduced by FastT$^-$ are almost the same, regardless of the type of underlying storage devices (i.e., UFS or eMMC). Therefore, the overall improvement ratio by FastT$^-$ becomes more significant for the fast storage, where FG apps generally exhibit shorter response times. This means that as the storage devices evolve in its speed, the effect of FastT becomes more substantial.

Even though FastT$^-$ gave FG I/Os the highest priority combined with a fast I/O path, we still observed that FastT$^-$ showed longer response times than FG-Only in all the scenarios. When we compare Fig. 11(a) and Fig. 6(a), the throughput of FG I/Os was not improved as much as FG-Only. This is because FastT$^-$ cannot resolve the I/O-priority inversion problem inside the storage device. Because of a priority-unaware I/O scheduler, for example, FG writes are always delayed by BG reads.

5.2 Performance Analysis on Emulator

Although the evaluation results in Section 5.1 showed that FastT$^-$ is quite effective on real smartphones, FastT$^-$ didn't reveal the full potential of our proposed FastT because it cannot fully control the storage device-internal I/O scheduler. In order to better understand the real effect of FastT on user experience, it is important to implement the proposed I/O device scheduler (in Section 4.5) with a complete support for the fast I/O path from the Android platform to the storage device. Since storage vendors do not allow to modify their firmware inside their storage devices, we performed evaluations using an emulated storage device with I/O traces collected from real smartphone apps.

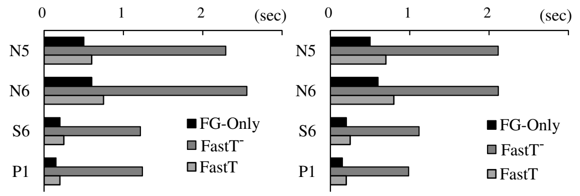


(a) Camera under Backup with FastT⁻.

(b) Camera under Backup with FastT.

Fig. 11: Storage-level snapshot of FG I/Os and BG I/Os.

We have implemented an emulation layer on top of an off-the-shelf SSD to emulate I/O latency and throughput of eMMC and UFS devices. This work is done by using a storage emulator developed for our prior studies [14]. Then, we collected the app status information, along with I/O traces, while executing scenarios described in Section 5.1 on the smartphone. The collected I/O traces were replayed on the emulated storage. Finally, we implemented the device I/O scheduler on the emulated storage which processed FG I/Os with a higher priority.



(a) Camera under Update. (b) Camera under Backup.

Fig. 12: Response time analysis on emulator.

We compared the performance of three policies: FG-only, FastT^{-}, and FastT, where FastT is FastT^{-} with the device I/O scheduler. Fig. 12 shows that FastT greatly improved the performance of Camera under both Update and Backup. In the case of Camera under Update on N6, FastT^{-} achieved the response time of 2.53 seconds, whereas FastT achieved the response time of 0.75 seconds, which is quite close to 0.6 seconds of FG-Only. For Camera under Backup, similarly, FastT achieved an equivalent response time to FG-Only. Compared with Fig. 11(a), in Fig. 11(b), we observe that FG I/Os were processed at a much higher throughput with negligible delays at the storage device level. This result shows that higher performance can be achieved if the storage device is able to handle I/O requests with the app-level priority information.

Finally, Fig. 13 shows how FastT scales when the number of BG apps increases from two to six. In addition to Update and Backup, we used four more I/O-intensive BG apps for this experiment. As shown in Fig. 13, the normalized app switch time increases from 1.1 to 1.27 only as the number of BG apps increases from 2 to 6. These results indicate that FastT can provide the equivalent level of responsiveness to an FG app regardless of the number of BG apps running with the FG app. Fig. 13 also shows that FastT is effective in improving the app switch time over the existing Android implementation (indicated by FG+BG), reducing the app switch time by 94% when 6 BG apps run.

6 Related Work

Various I/O scheduling techniques have been proposed to address the problem caused by BG I/Os [18, 19, 49, 50]. A boosting quasi-async I/O (QASIO) is one of such efforts to provide better I/O scheduling by means of the priority inheritance protocol [18]. QASIO is motivated by an observation that high-priority sync writes are often delayed by low-priority async writes. QASIO improve overall I/O responsiveness by temporarily increasing the priority of async writes over sync ones. A request-centric I/O prioritization (RCP) [19] is proposed which is also based on the priority inheritance protocol. RCP further improves QASIO by prioritizing I/O requests at the page cache layer rather than the block I/O layer.

While still effective, both QASIO and RCP have fundamental limitations in improving I/O responsiveness, in comparison to FastTrack. First, both techniques are not aware of FG I/Os and BG I/Os in smartphones, and thus, they are unable to prioritize FG I/Os that have a high impact on user-perceived response times. Second, QASIO and RCP both rely on the priority inheritance protocol. Thus, they cannot remove additional delays required for high-priority I/Os to wait until low-priority ones finish. Therefore, their effectiveness on improving user-perceived latency is limited on highly interaction-oriented devices like smartphones.

Foreground app-aware I/O management (FAIO) [49] is the first technique that accelerates FG I/Os by adopting I/O preemption in smartphones. FAIO analyzes FG app information to identify FG I/Os and preempts BG I/Os to quickly process FG I/Os. However, since FAIO uses I/O preemption only at the page cache level, it does not resolve the priority inversion problem at the storage device level. It also fails to prevent performance degradation caused by the aggressive evictions of FG data from page cache under BG I/O intensive workloads.

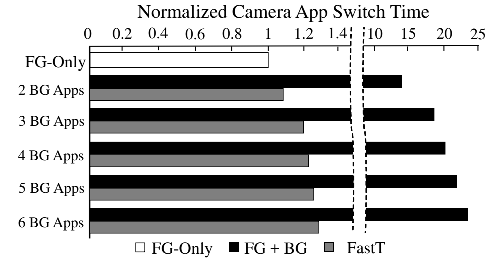


Fig. 13: Scalability of FastT over the varying BG apps.

7 Conclusions

We have presented a foreground app-aware I/O management scheme, FastTrack, which significantly improves the quality of user experience on smartphones by avoiding I/O-priority inversions between a foreground app and background apps on Android smartphones. Unlike the existing techniques, FastTrack employs a preemption-based approach for fast responsiveness of a foreground app. In order to support I/O-priority-based preemption in a holistic fashion, FastTrack reimplemented the page cache in Linux and the storage-internal I/O scheduler which previously operated in a foreground app-oblivious fashion. From a systematic analysis study, these two modules were identified as the root causes of most I/O-priority inversions. Our experimental results on real smartphones show that FastTrack is effective in improving the quality of user experience on smartphones. For example, FastTrack achieved the equivalent quality of user experience of a foreground app regardless of the number of concurrent background apps.

FastTrack can be extended in several directions. For example, we believe that our preemption-based approach can be extended to other computing environments where a strong requirement on the response time exists.

8 Acknowledgments

We would like to thank anonymous referees for valuable comments that greatly improved our paper. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (NRF-2015M3C4A7065645 and NRF-2018R1A2B6006878). The ICT at Seoul National University provided research facilities for this study. Sungjin Lee was supported by the NRF grant funded by the Korea government (Ministry of Science and ICT) (NRF-2017R1E1A1A01077410) and the DGIST R&D Program of the Ministry of Science and ICT (18-EE-01). (*Corresponding Author: Jihong Kim*)

References

[1] Android Performance Management. https://source.android.com/devices/tech/power/performance.

[2] Android Process. https://developer.android.com/reference/android/os/Process.html.

[3] KWON, Y., LEE, S., YI, H., KWON, D., YANG, S., CHUN, B., HUANG, L., MANIATIS, P., NAIK, M., AND PAEK, Y. Mantis: automatic performance prediction for smartphone applications. In *Proceedings of the USENIX Conference on Annual Technical Conference* (2013).

[4] MITTAL, T., SINGHAL, LOKESH., AND SETHIA, D. Optimized CPU Frequency Scaling on Android Devices Based on Foreground Running Application. In *Proceedings of the Fourth International Conference on Networks and Communications* (2013).

[5] SONG, W., SUNG, N., CHUN, B., AND KIM, J. Reducing Energy Consumption of Smartphones Using User-Perceived Response Time Analysis. In *Proceedings of the International Workshop on Mobile Computing Systems and Applications* (2014).

[6] Optimizing Foreground App Performance on Nexus S. https://www.reddit.com/r/Android/comments/1wqcuh/how_do_i_make_android_manage_foreground_apps.

[7] RAM Issue on Nexus S. https://forum.xda-developers.com/nexus-s/help/ram-issues-nexus-s-jelly-bean-t1854513.

[8] Performance Drop on Nexus S. http://forums.whirlpool.net.au/archive/1999853.

[9] The First Android Smartphone in the World with 8 GB of RAM. http://bgr.com/2017/01/05/asus-zenfone-ar-release-date.

[10] Asus ZenFone AR. https://www.asus.com/us/Phone/ZenFone-AR-ZS571KL.

[11] Android Mobile Phones with 6 GB RAM. https://www.techmanza.in/6gb-ram-mobile.html.

[12] Nexus S. https://en.wikipedia.org/wiki/Nexus_S.

[13] Samsung Galaxy S8. https://en.wikipedia.org/wiki/Samsung_Galaxy_S8.

[14] HAHN, S.S., LEE, S., JI, C., CHANG, L., YEE, I., SHI, L., XUE, C.J., AND KIM, J. Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter In *Proceedings of the USENIX Annual Technical Conference* (2017).

[15] KIM, H., LEE, M., HAN, W., LEE, K., AND SHIN, I. Aciom: Application Characteristics-aware Disk and Network I/O Management on Android Platform. In *Proceedings of the International Conference on Embedded Software* (2011).

[16] VALENTE, P., AND ANDREOLINI, M. Improving application responsiveness with the BFQ disk I/O scheduler. In *Proceedings of the 5th Annual International Systems and Storage Conference* (2012).

[17] NGUYEN, D. Improving smartphone responsiveness through I/O optimizations. In *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous* (2014).

[18] JEONG, D., LEE, Y., AND KIM, J. Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2015).

[19] KIM, S., KIM, H., LEE, J., AND JEONG, J. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2017).

[20] Priority Inheritance. https://en.wikipedia.org/wiki/Priority_inheritance.

[21] Nexus 5. https://en.wikipedia.org/wiki/Nexus_5.

[22] Nexus 6. https://en.wikipedia.org/wiki/Nexus_6.

[23] Samsung Galaxy S6. https://en.wikipedia.org/wiki/Samsung_Galaxy_S6.

[24] Pixel. https://en.wikipedia.org/wiki/Pixel_(smartphone).

[25] Dropbox. https://en.wikipedia.org/wiki/Dropbox_(service).

[26] OneDrive. https://en.wikipedia.org/wiki/OneDrive.

[27] KUMAR, U. Understanding Android's Application Update Cycles. https://www.nowsecure.com/blog/2015/06/08/understanding-android-s-application-\\update-cycles.

[28] Twitter Version History. https://www.apk4fun.com/history/2699.

[29] DRAGO, I., MELLIA, M., MUNAFO, M.M., SPEROTTO, A., SADRE, R., AND PRAS, A. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the Internet Measurement Conference* (2012).

[30] QuickPic Gallery. https://play.google.com/store/apps/details?id=com.alensw.PicFolder.

[31] Android Camera API. https://developer.android.com/guide/topics/media/camera.html.

[32] Dragon Ball Z Dokkan Battle. https://play.google.com/store/apps/details?id=com.bandainamcogames.dbzdokkanww.

[33] strace - trace system calls and signals. https://linux.die.net/man/1/strace.

[34] Completely Fair Queueing. https://en.wikipedia.org/wiki/CFQ.

[35] Embedded MultiMediaCard (e.MMC). http://www.jedec.org/standards-documents/technology-focus-areas/flash-memory-ssds-ufs-emmc/e-mmc.

[36] Universal Flash Storage (UFS). http://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs.

[37] Physical Page Allocation. https://www.kernel.org/doc/gorman/html/understand/understand009.html.

[38] Blocking I/O. http://www.makelinux.net/ldd3/chp-6-sect-2.

[39] Memory Mapping and DMA. https://static.lwn.net/images/pdf/LDD3/ch15.pdf.

[40] Android App Dependency Configuration. https://developer.android.com/studio/build/dependencies.html.

[41] Predefined UIDs for Android Processes. https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/Process.java.

[42] Atomic context and kernel API design. https://lwn.net/Articles/274695/.

[43] JEONG, S., LEE, K., SON, S., AND WON, Y. I/O Stack Optimization for Smartphones. In *Proceedings of the USENIX Conference on Annual Technical Conference* (2013).

[44] NAM, E.H., KIM, B.S.J., EOM, H., AND MIN, S.L. Ozone (O3): An Out-of-Order Flash Memory Controller Architecture. *IEEE Transactions on Computers* (2013), vol. 60, pp. 653-666.

[45] HAHN, S.S., LEE, S., AND KIM, J. SOS: Software-based out-of-order scheduling for high-performance NAND flash-based SSDs. In *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies* (2013).

[46] JUNG, M., CHOI, W., SHALF, J., AND KANDEMIR, M.T. Triple-A: a Non-SSD based autonomic all-flash array for high performance storage systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2014).

[47] SCSI command. https://en.wikipedia.org/wiki/SCSI_command.

[48] Hearthstone. https://play.google.com/store/apps/details?id=com.blizzard.wtcg.hearthstone.

[49] HAHN, S.S., LEE, S., YEE, I., RYU, D., AND KIM, J. Improving User Experience of Android Smartphones Using Foreground App-Aware I/O Management. In *Proceedings of the Asia-Pacific Workshop on Systems* (2017).

[50] JAUHARI, R., CAREY, M.J., AND LIVNY, M. Priority-Hints: An Algorithm for Priority-Based Buffer Management. In *Proceedings of the International Conference on Very Large Data Bases* (1990).

Mainstream: Dynamic Stem-Sharing for Multi-Tenant Video Processing

Angela H. Jiang, Daniel L.-K. Wong, Christopher Canel, Lilia Tang, Ishan Misra,
Michael Kaminsky[†], Michael A. Kozuch[†], Padmanabhan Pillai[†],
David G. Andersen, Gregory R. Ganger
Carnegie Mellon University; [†]Intel Labs

Abstract

Mainstream is a new video analysis system that jointly adapts concurrent applications sharing fixed edge resources to maximize aggregate result quality. Mainstream exploits partial-DNN (deep neural network) compute sharing among applications trained through transfer learning from a common base DNN model, decreasing aggregate per-frame compute time. Based on the available resources and mix of applications running on an edge node, Mainstream automatically determines at deployment time the right trade-off between using more specialized DNNs to improve per-frame accuracy, and keeping more of the unspecialized base model to increase sharing and process more frames per second. Experiments with several datasets and event detection tasks on an edge node confirm that Mainstream improves mean event detection F1-scores by up to 47% relative to a static approach of retraining only the last DNN layer and sharing all others ("Max-Sharing") and by 87X relative to the common approach of using fully independent per-application DNNs ("No-Sharing").

1 Introduction

Video cameras are ubiquitous, and their outputs are increasingly analyzed by sophisticated, online deep neural network (DNN) inference-based applications. The ever-growing capabilities of video and image analysis techniques create new possibilities for what may be gleaned from any given video stream. Consequently, most raw video streams will be processed by multiple analysis pipelines. For example, a parking lot camera might be used by three different applications: reporting open parking spots, tracking each car's parking duration for billing, and recording any fender benders.

This paper focuses on video processing on edge devices, which will be a common way to address bandwidth limitations, intermittent connectivity (e.g., in drones), and real-time requirements. Applications executing at the edge, though, face tighter bounds on resource availability than in datacenters. Naturally, optimal video application performance requires tuning for the resources available [48, 12, 51, 18, 26].

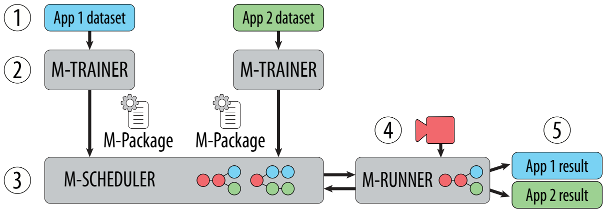


Figure 1: Mainstream Architecture. Offline, for each task, M-Trainer takes a labeled dataset and outputs an M-Package. M-Scheduler takes independently generated M-Packages, and chooses the task-specific degree of specialization and frame rate. M-Scheduler deploys the unified multi-task model to M-Runner, performing inference on the edge.

Unfortunately, what resources will be available to the application at deployment time is often unknown to the developer. Further, resource availability changes as additional applications arrive and depart. Instead, individual application developers typically develop their models in isolation, assuming either infinite resources or a predetermined resource allotment. When a number of separately tuned models are run concurrently, resource competition forces the video stream to be analyzed at a lower frame rate—leading to unsatisfactory results for the running applications, as frames are dropped and events in those frames are missed. However, due to the popularity of *transfer learning* (Sec. 2) [40, 47, 37, 43], contention can be reduced by eliminating redundant computation between concurrent applications [18].

Mainstream is a new system for video processing that addresses resource contention by dynamically tuning degrees of work sharing among concurrent applications. Specifically, it focuses on sharing portions of DNN inference, which consumes the majority of video processing cycles. Mainstream exploits the potential "shared stem" of computation that results from application developers' use of the standard DNN training approach of transfer learning. In transfer learning, training begins with an existing, *pre-trained* DNN, which is then re-trained for a different task. Typically, only a subset of the pre-trained DNN is specialized; when different applications start with the same pre-trained DNN, Mainstream identifies the common layers and executes them only once per frame.

A critical challenge of exploiting shared stems well is determining how much to share. Application developers usually specialize as much of the pre-trained DNN as is necessary to achieve high model accuracy. More specialization, however, means that less of the pre-trained DNN can be shared. Thus, there is an explicit trade-off between the benefits of greater per-frame accuracy (via more-specialized DNNs) and processing more frames of the input video stream (via more sharing of less-specialized DNNs). The right choice depends on the edge device resources, the number of concurrent applications, and their individual characteristics.

Deployment time model selection. Mainstream moves the final DNN model selection step from application development time to deployment time, when the hardware resources and concurrent application mix are known. By doing so, Mainstream has the necessary information to select the right amount of DNN specialization (and thus sharing) for each application. As applications come and go, Mainstream can dynamically modify its choices. Previous systems like VideoStorm [48] select models by considering each application independently. *The specialization-vs-sharing trade-off, however, can only be optimized when considering applications jointly.* Joint optimization produces a combinatorial set of options, which Mainstream navigates using application metadata and domain-specific models; the system uses this information to estimate the effects of DNN specialization and frame sampling rate on application performance. Unlike black-box approaches, Mainstream can jointly optimize for stem-sharing without needing to profile each combination.

Mainstream consists of three main parts (Fig. 1). The *M-Trainer* toolkit helps application developers manage their training process to produce the information needed to allow tuning the degree of specialization at deployment time. Current standard practice is for developers to experiment with different model types, hyperparameters, and degrees of re-training to find the best choice for an assumed resource allocation, discarding the trained DNN models not chosen. M-Trainer instead keeps "less optimal" *candidate* DNN models, together with associated training-time information (e.g., per-frame accuracy, event detection window). The *M-Scheduler* uses this information, together with a profile of per-layer runtime on the target edge device, to determine the best candidate for each application—including the degrees of specialization and, thus, sharing. It efficiently searches the option space to maximize application quality (e.g., average F1 score among the applications). The *M-Runner* runtime system runs the selected DNNs, sharing the identical unspecialized layers.

Experiments with several datasets and event detection tasks on an edge node confirm the importance of making deployment-time decisions and the effectiveness of Mainstream's approach. Results show that dynamic selection of shared stems can improve F1-scores by up to 87X relative to the common approach of using fully-independent per-application DNNs (No-Sharing) and up to 47% compared to a static approach of retraining only the last DNN layer and sharing all others (Max-Sharing). Across a range of concurrent applications, Mainstream adaptively selects a balance between per-frame accuracy and frame sampling rate that consistently provides superior performance over such static approaches.

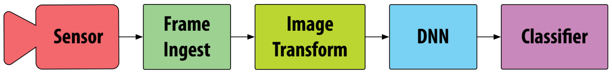


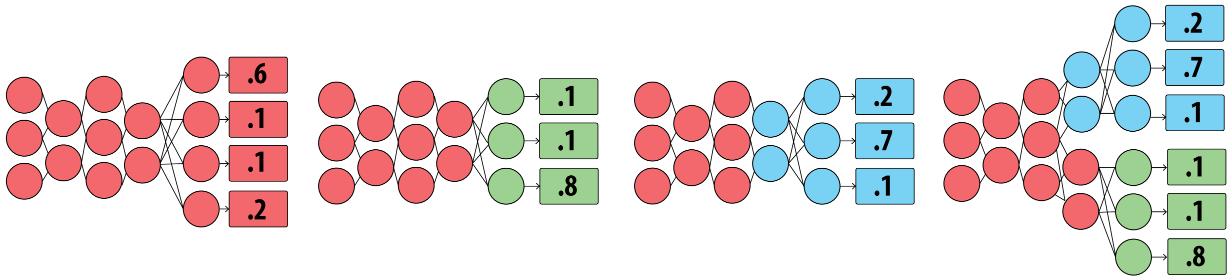
Figure 2: Example computation pipeline for event detection.

Contributions. This paper makes three main contributions. First, it highlights the critical importance of reducing aggregate per-frame CPU work of multiple independently developed video processing applications via stem-sharing; No-Sharing is unable to support even three concurrent applications on our edge node deployment. Second, it identifies the goodness trade-off between per-frame quality and the frame sampling rate dictated by the degree of DNN specialization (and thus the amount of sharing). Third, it describes and demonstrates the efficacy of the Mainstream approach for automatically deploying the right degree of specialization for each submitted application's DNN.

2 Background

DNNs are a powerful tool used in computer vision tasks such as human action recognition [43], object detection [15], scene geometry estimation [14], face recognition [45], etc. Fig. 2 shows a typical computation pipeline for an image classification application. Although frame ingest and image preprocessing are necessary stages of computation, they are low cost and easily shared between concurrent applications. DNNs, on the other hand, are typically unique to each application and computationally expensive: in one image classification application we run, the DNN inference incurs 25X more latency than the preprocessing steps.

DNNs and transfer learning. A machine learning (ML) *model* is a parameterized function that performs a task. *Training* is the process of learning parameter values (called weights) such that the model will approximate the desired function with some measure of accuracy. For example, when training an image classifier, one might examine labeled input images and use gradient descent to find a set of weights that minimizes a loss function over the labels. Using the trained model to find the function's



(a) Base DNN model **(b)** App. #1's DNN model **(c)** App. #2's DNN model **(d)** As executed in Mainstream

Figure 3: Fig. 3a depicts a base DNN trained from scratch for its task. Fig. 3b and Fig. 3c show two new task DNNs, fine-tuned against the base DNN. App. #1 freezes more layers during training than App. #2. Fig. 3d shows how Mainstream runs the applications concurrently. Layers frozen by both App. #1 and App. #2 can be shared.

Architecture	Number of Layers	ImageNet Top-1 Accuracy (%)
InceptionV3	314	78.0
MobileNets-224	84	70.7
ResNet-50	177	75.6

Table 1: Top-1 accuracy of three neural networks architectures trained on the ImageNet dataset.

output given a new, unlabeled input is called *inference*.

DNNs are a class of ML models that usually have a large input space, such as the pixels of an image. A DNN can be represented by a graph where nodes are organized into *layers*; each node computes a function of its inputs, which are outputs from the previous layer.

The "deep" in DNNs refers to their many layers. Increasingly, successful applications of DNNs have largely been the result of building models with more layers that take larger vectors of inputs [42, 29, 19, 44]. The success of these models has hinged critically upon the arrival of very large, labeled datasets for training [13, 32, 3].

Training these large models is notoriously hard. One often lacks sufficient labeled data or computational resources to train such a model. *Transfer learning* is an alternative to training a model from scratch. Here, a model that has already been trained on a similar task (a *base DNN* as in Fig. 3a), is used as an initialization point or feature extractor for the new *target DNN*. During training, a subset of the old parameters are *frozen* and do not change. The remaining free parameters are then retrained on the new task with a new training dataset (Fig. 3b and Fig. 3c). This process *fine-tunes* these parameters to achieve a result comparable to end-to-end training on the entire DNN, but does so with much less data and at a much lower computational cost. In practice, few practitioners train networks from scratch, let alone develop novel network architectures. Transfer learning via one of a few popular neural networks is standard practice.

DNNs for image classification and event detection. Although we believe that Mainstream's approach is generally applicable to video stream analysis, in this paper, we focus on applications that use image-classification DNNs to perform event detection.

Image classification aims to assign one label from a set of categories or *classes* to each image. For example, a 10-class classifier takes an input image and returns a 10-item vector of probabilities representing the likelihood that the image belongs to each class. *Top-N accuracy* is the probability that the correct label is among the top-N highest probability output labels. So, Top-1 accuracy indicates the fraction of images that the model classifies correctly. We refer to this metric as the *per-frame accuracy* in the context of video classification. Popular neural network architectures for image classification include ResNet [19], InceptionV3 [44], and MobileNets [20]. Table 1 describes these three neural networks and their Top-1 accuracy achieved on the ImageNet dataset [13]. Networks trained on ImageNet are popular base-DNNs for image classification tasks.

We define an event as a contiguous group of frames containing some visible phenomenon that we are trying to identify: e.g., a cyclist passing by, or a puff of smoke being emitted. One way of doing event detection is to perform image classification across a sequence of frames. An event is detected if at least one of the contiguous frames is sampled, analyzed, and correctly labeled. Previous works [31] have also used this existence metric to measure recall and precision of range-based queries. (Event detection is not to be confused with object detection, where the goal is to locate an object in a single frame. Indeed, object detection is another way of performing event detection.) We evaluate event classification applications by measuring the *event F1-score*, the harmonic mean between *event recall* and *event precision*. The event recall reports the proportion of ground truth events identified. The event precision reports the proportion of classified events that are true positives. Note that these metrics are relative to the detection of events across multiple frames and are distinct from per-frame metrics (e.g., Top-1 accuracy).

3 Mainstream Approach: What and Why

Sharing computation between DNNs. When supporting multiple inference applications on a single infrastructure, the common approach is to execute every application's DNN model independently. We refer to this as a "No Sharing" approach. To avoid redundant work, Mainstream instead computes results for DNN layers common to multiple concurrent applications just once and distributes the outputs of shared stems to the specialized layers of all sharing applications. This is analogous to common subexpression elimination used in other domains, e.g., optimizing compilers or database query planners.

Fig. 3 illustrates how compute sharing can be leveraged when two DNNs are fine-tuned from a common pre-trained model and have some unspecialized layers in common. Compute sharing can significantly affect per-frame computation cost and improve throughput for a given CPU resource. Fig. 4a quantifies this effect. It shows the throughput achieved by Mainstream running up to eight concurrent InceptionV3-based event detection pipelines, as a function of how many DNN layers they have in common (i.e., their common degree of specialization). With no sharing (the left-most points), adding a second application halves throughput, which continues to degrade geometrically as more applications are added. Moving to the right, throughput improves as more layers are shared. When all but the last layer are shared, additional applications can be run at very low marginal cost.

On the other hand, there are costs to enabling sharing by leaving many layers unspecialized. In particular, the per-frame accuracy of a model may be lower when only a few layers are specialized. Fig. 4b shows the effect of specialization on per-frame accuracy for several combinations of DNN architectures and classification tasks. As expected, accuracy decreases slowly as less-specialized networks are employed (and hence more sharing is enabled)— with a large decrease occurring only when the fraction of the network specialized is very small. This characteristic enables Mainstream to share large portions of the network with low accuracy loss.

Adaptive management of the sharing opportunity. Since transfer learning is so commonly used by ML developers, and base models are shared within organizations and on the Internet, there may be many opportunities to exploit inter-DNN redundancy in the unspecialized layers. Most developers either use a popular default of specializing only the last layer (which is great for sharing potential, at the potential cost of model accuracy) or determine the degree of specialization based on the amount of training data available, since retraining too many layers without sufficient training data leads to over-fitting. Notably, each developer decides independently.

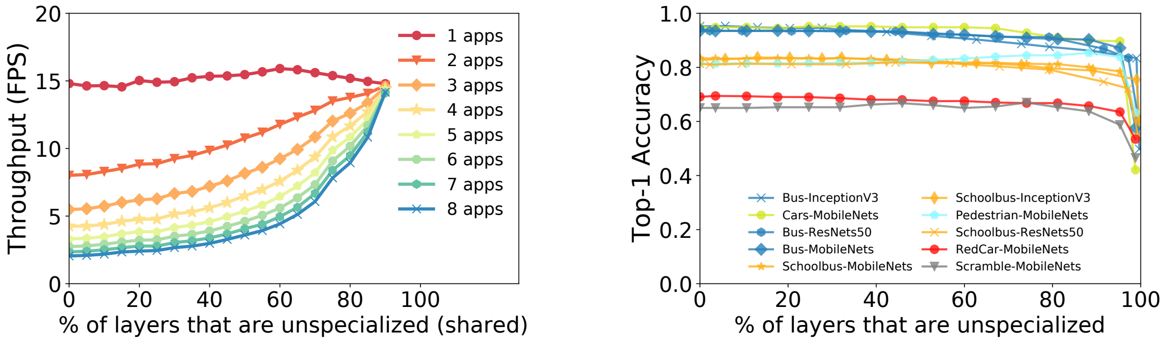
The problem with this approach is that the impact of sharing computation on application quality depends on factors only known at deployment time: the set of concurrent applications and the resources of the edge node on which they are run. Hence, Mainstream defers the decision regarding how much specialization to employ from application development time to deployment time.

Impact of sampling rate for event detection. Given the trade-off between per-frame accuracy and frame processing throughput, picking the right degree of specialization is challenging. Consider an application for monitoring environmental pollution from trains, which is being built using a train detector we deployed. When the application detects a train coming into view, it triggers the capture of high fidelity frames of the train's smoke stack (for subsequent pollution analysis).

Increasing specialization to improve per-frame accuracy increases the probability of correctly classifying frames containing trains— but reduces shared computation. This, in turn, leads to less frequent sampling, which removes opportunities to analyze frames containing a particular view of a train. A higher frame rate increases the probability that an event will be observed in more frames, creating more opportunities for detection. So, the question becomes: should one sample more frames using a less accurate model or sample fewer frames using a more accurate model?

Analytical model for event detection. The Mainstream scheduler (Sec. 6) navigates this "accuracy vs. sampling rate" space by evaluating various candidate *{specialization, frame rate}* tuples. To do so, however, the scheduler must be able to interpret the benefit at the application (not per-frame) level. Hence, we propose an analytical model (sketched in Equations 1-4, below) that approximates the *event* F1-score for a given DNN, given estimates of (a) event length, (b) event frequency, (c) the correlation between frames (discussed below), (d) per-frame DNN accuracy, and (e) DNN analysis frame rate. The frame rate (e) comes directly from the scheduler's proposed tuple; similarly, the accuracy (d) associated with a given specialization proposal will be available to the scheduler (see Sec. 5). Values for event length (a), frequency (b), and correlation (c) can either be measured using representative video samples, or they can be estimated by the developer.

We are able to predict the application's F1-score by estimating the expected number of frames per event that we will have the opportunity to analyze and computing the probability that analyzing the set of frames will result in a detection. The expected number of frames analyzed per event is dependent on the event length and frame rate. The per-frame Top-1 accuracy represents the probability that



(a) Throughput vs. Sharing

(b) Per-frame accuracy vs. (Potential) sharing

Figure 4: Conflicting consequences of DNN compute sharing. (a) shows the frame processing rate for 1–8 concurrent event detection applications as a function of the fraction of the InceptionV3 DNN they share, from No-Sharing on the left to sharing all but the last layer (Max-Sharing) on the right. The experiments are run on the hardware described in Section 7. (b) shows Top-1 accuracy as a function of the fraction of unspecialized layers for three popular DNN architectures (ResNet-50 [19], InceptionV3 [44], and MobileNets-224 [20]) using six of the datasets described in Table 2. We trained each classifier using all three network architectures but omitted some curves for brevity. The horizontal axis starts from fully specialized DNNs on the left to only the last layer being specialized on the right; recall that potential computation sharing is limited to the unspecialized layers.

we will classify any individual frame correctly. However, this does not factor in the fact that sequential frames of an event may be correlated in some way. We therefore introduce and estimate the *inter-frame correlation*, which measures the marginal benefit of analyzing more frames of a single event.

The inter-frame correlation, *corr*, is based on conditional probabilities. For frames corresponding to an event, if P(X_i) and P($\neg X_i$) are the probabilities of detecting or not detecting the event in frame i, respectively, then $P(\neg X_2 \mid \neg X_1)$ is the probability of not detecting the event in frame X_2, given that we did not detect it in frame X_1. This conditional probability can be measured empirically from training data. Relying on the Kolmogorov definition, we can calculate the probability the event is detected in at least one of the two frames as $1 - P(\neg X_2 \mid \neg X_1) * P(\neg X_1)$. This logic can be extended to approximate the probability of detecting the event in N tries and to estimate recall.

To estimate recall, we calculate the probability that our DNN will correctly classify at least one frame of an event using the following steps:

$$N = \begin{cases} \left\lceil \frac{d}{stride} \right\rceil & \text{w.p. } \frac{d-(stride\%d)}{stride} \\ \left\lfloor \frac{d}{stride} \right\rfloor & else \end{cases} \tag{1}$$

$$P_{miss\_1} = 1 - accuracy_{per-frame} \tag{2}$$

$$P_{miss\_N} = corr^{N-1} * P_{miss\_1} \tag{3}$$

$$recall = 1 - P_{miss\_N} \tag{4}$$

We use Eq. 1 to calculate N, the expected number of frames of the event that the model will process. Here, d is the event length, and *stride* is the inverse of the frame rate. Equation 3 estimates the probability the DNN misclassifies all N analyzed frames. Recall is the complement: the probability that we correctly classify at least one frame of the event.

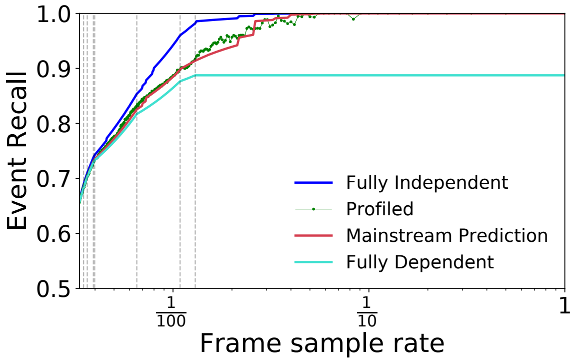


Figure 5: Effect of sample-rate on recall, for different inter-frame correlations. The dotted vertical lines represent each train in the dataset, denoting $\frac{1}{trainlength}$, which is the sample rate required to ensure that one frame of that train is analyzed. The "Profiled" line is measured directly from the TRAIN video dataset, and the other three are approximations based on different inter-frame correlations (uncorrelated, fully correlated, and the empirical correlation observed in the TRAIN dataset).

To estimate the false positive rate, we repeat this calculation, except that d is the number of frames between events (derived from the event frequency), and P_{miss_1} is the probability of *true* negatives. The true positive rate, the false positive rate, and the event frequency are used to calculate the precision. The F1-score is the harmonic mean between precision and recall.

To evaluate our model, we profile an application and measure the actual recall observed when running the event detector application (e.g., train detection) on the video stream at different sampling rates (Fig. 5). This was measured by averaging the recall achieved from 10,000 trials of sampling at each sample rate. The result is plotted by the "Profiled" line. Our analytical model ("Mainstream Prediction") is sufficiently accurate to describe the complex relationship between frame sample rate and application recall. Mainstream uses this analytical model to efficiently optimize for the trade-off between per-frame accuracy and frame rate.

We use Mainstream for event detection but believe its approach can be generally applied to DNN-based tasks where application quality depends not only on its model, but also on its input sampling rate (e.g., object tracking, action recognition.)

4 Mainstream Architecture

We have developed Mainstream, a training and runtime system for DNN-based live analytics of camera streams, which (a) enables efficient sharing of computation between detection applications, (b) maximizes event F1-score across all tasks, and (c) allows each task to be independently developed, trained, and deployed. Fig. 1 shows the architecture of Mainstream, which consists of three components: M-Trainer, M-Scheduler, and M-Runner.

To deploy a new application to Mainstream, the user provides a corresponding labeled training dataset to their local instance of M-Trainer (Step 1). M-Trainer uses the dataset to train a number of potential models, with varying numbers of specialized layers. This *Model Set* and associated metadata are then assembled into an "M-Package" (Step 2). Note that these are offline steps, performed just once per application prior to deployment, independent of all other tasks. At runtime, M-Scheduler uses the M-Packages of all currently-deployed applications to determine, for each application, the degree of DNN sharing and sampling rate such that, across all applications, the event F1-score is maximized, subject to the resource limits of the edge platform (Step 3). M-Scheduler runs in the datacenter, and is executed once for each scheduling event (e.g., a change in the deployed set of applications, or in available hardware resources). M-Runner then executes the selected model configuration on edge devices (Step 4) and returns app-specific results in real-time (Step 5).

M-Runner is a relatively straightforward execution system for running visual processing pipelines. It accepts a DAG, where each node represents a unit of independent computation, and connections represent data flow. Fig. 2 illustrates the logical DAG for an image classification application. Most of the computation is expected in the "DNN" process, which evaluates the merged DNN of all concurrent tasks. This combined DNN, as illustrated in Fig. 3d, represents the set of models selected by M-Scheduler across all tasks. This DNN is structured as a tree, with sets of layers branching from the shared stem. M-Runner executes the shared stem once per frame, reducing the total processing costs of the deployed tasks.

We next describe how M-Trainer independently trains model candidates for potential sharing (Sec. 5) and how M-Scheduler dynamically chooses among them (Sec. 6).

5 Distributed Sharing-Aware Training

M-Trainer produces a set of models for each task so that they can be combined dynamically at runtime to maximize collective performance. Application developers use M-Trainer independently at different times and locations.

One approach to sharing computation between applications would be to jointly train them using a multi-task network. This, however, requires centralized training of applications. MCDNN [18] proposed fine-tuning models independently and sharing the unspecialized DNN layers. This static approach prevents M-Scheduler from dynamically tailoring stem-sharing to the available resources. In contrast, M-Trainer generates a *set of models* that vary in the number of specialized layers. These models compose an application-specific *Model Set*. The generation of Model Sets allows for the late binding of the degree of specialization to deployment time, when the platform characteristics and co-deployed applications are known.

To construct a Model Set, M-Trainer first analyzes the structure of the base DNN to identify *branchpoints*, the potential boundaries between frozen and specialized layers. Using the app-specific training data provided by the developer, M-Trainer generates a set of fine-tuned DNN models, one per branchpoint, where layers up to the branchpoint are frozen, and the rest are specialized. Only the models at the Pareto-optimal frontier with respect to number of layers specialized and estimated accuracy are actually included in the M-Package. This eliminates from consideration models that reduce accuracy, while requiring more specialization. For example, an overfitted model, caused by specializing too many layers with insufficient training data, will not be included.

Model Sets are bundled together with application metadata into an *M-Package*. This metadata includes the measured per-frame accuracy of each model (we use a portion of the data as the validation dataset.) The expected minimum event duration, event frequency, and inter-frame correlation are optionally measured from the training data and included in the M-Package, or directly provided by the application developer.

The construction of the M-Package is an offline operation, which is run just once per application. For each application, M-Trainer must train multiple models. Although training a model from scratch can be very resource intensive, fine-tuning is much quicker. M-Trainer creates Model Sets with 15 model options in 8 hours on a single GPU (Sect. 7). Note that the computation for generating all of the models is easily parallelized in a datacenter setting, and may not be significantly higher than traditional fine-tuning. For example, to find the right number of layers to specialize in order to maximize accuracy, one may need to generate these models anyway. The key difference here is that intermediate runs are not discarded, and the final selection is made at run time by M-Scheduler.

As each application's models are independently trained and analyzed, no coordination or sharing of training data is needed between developers of different tasks. The resulting M-Package, however, contains enough information that M-Scheduler, at run time, can optimize across the independently-developed tasks.

6 Dynamic Sharing-Aware Scheduling

At each *scheduling event* (typically, an application submission or termination), M-Scheduler uses the M-Packages created by the various per-application M-Trainers to produce a new overall schedule that optimizes some global objective function, subject to resource constraints (currently, M-Scheduler maximizes average event F1-score across applications). The *schedule* consists of a DNN model selection (indicating the degree of sharing) and target frame-rate for every running application.[1] The final schedule is a tree-like model with applications splitting from a shared stem at potentially different branch points, with each application able to run at its own frame rate.

M-Scheduler optimization algorithm. With N applications to schedule, S possible specialization settings per application, and R frame-rate settings per application, the number of possible schedules is $(S \cdot R)^N$. Although this space is large (e.g., $N \approx 10$, $R \approx 10$, and $S \approx 10$ in our experiments), M-Scheduler can efficiently determine a good schedule using a greedy heuristic (Algo. 1). We compare the schedules generated by our greedy scheduler to those of an exhaustive scheduler in >4,800 workloads each consisting of up to 10 applications, and find that the greedy schedules are on average within 0.89% of optimal.

Essentially, at each step of our iterative algorithm, the scheduler considers making a *move* which improves the application quality of a single application by tweaking its frame rate and/or model specialization. The algorithm greedily selects the move that yields the best ratio of *benefit* to *cost*, defined below. Naturally, before this iterative refinement, the schedule is initialized to the lowest cost configuration— Max-Sharing with minimum frame rate. At any iteration step, the number of possible moves is bounded by $S \cdot R \cdot N$. The total number of moves per invocation of the scheduler is similarly bounded by $S \cdot R \cdot N$, but in practice is much fewer as the set of potential moves that fit within the computational budget is exhausted.

Algorithm 1 Scheduler optimization algorithm

```
function GetNextMove(schedule)
    ▷ Finds change to schedule with the highest $\frac{benefit}{cost}$
function Schedule(budget)
    sched ← GetSchedule(max_sharing, min_fps)
    while True do
        next_move ← GetNextMove(sched)
        cost ← cost + GetCost(next_move)
        if cost < budget then
            sched ← UpdateSchedule(next_move)
        else return sched
```

Measuring the Benefit of a Move. The benefit associated with a move captures the improvement in F1-score for the application associated with that move. This value is calculated using the analytical model presented in Sect. 3 and the application metadata in the M-Package.

Measuring the Cost of a Move. The cost value considered represents the computational resources (e.g., CPU-seconds per second) consumed by a given schedule arrangement and depends on the number of shared subgraphs, the number of task-specific subgraphs, and the intended throughput (frame-rate) of each subgraph. The relative cost of a schedule is the sum of the execution time of each model layer, multiplied by the desired throughput. Consider for example a schedule with two applications, both executing at F FPS. Assume they share a compute stem A, and then branch to specialized subgraphs B_1 and B_2. If C_A represents the execution cost (in CPU-seconds per frame, say) of A, and C_B the execution cost of B_1 and B_2, then the total cost of the schedule is $F \cdot C_A + 2F \cdot C_B$. Adding a third application based on the same network, using the same branchpoint and frame rate will add another factor of $F \cdot C_B$ to the schedule cost.

To most accurately model the compute costs (e.g., C_A and C_B), a forward pass execution of the base DNN should be executed and measured once on the target hardware. Note that as cost is relatively insensitive to the assigned model weights, each base DNN need only run one time (ever) per target hardware, not once per trained application.

Max-Min Fairness Among Applications. Although stem-sharing improves overall system efficiency, maximizing a global objective may lead to an inequitable allocation of resources for individual applications. Thus, M-Scheduler can also be run in max-min fairness mode, which maximizes the minimum benefit among applications, instead of the average. Max-min fairness is scheduled by searching the space using dynamic programming.

[1] Here, we assume that some admission control policy (outside the scope of this work) has been applied to ensure that some schedule is feasible for the set of running applications.

X-Voting To Improve Precision. Mainstream uses *voting* across frames to improve precision, and consequentially F1-score. With X-voting, Mainstream requires X consecutive positives to identify an event. While X-voting decreases false positives, it is not guaranteed to increase precision. For X-voting to improve precision, it must decrease false positives at a higher rate than true positives. The ideal X-voting configuration again depends on the applications and the resources available. A higher *X* incurs fewer false positives, but requires more cost to sustain high recall (either by increasing FPS or increasing specialization). We evaluate the effect of various X-voting configurations in Sect. 8.

7 Experimental Setup

To evaluate our system, we implement seven different event detection applications. We refer to the set of applications as 7-Hybrid. These are listed in Table 2, along with the datasets we used to train and test them. A pedestrian-detection application (Pedestrian) is trained based on the fully-labeled, publicly-available Urban Tracking video dataset [25]. Our application to classify car models (Car) uses the Stanford Cars image dataset [28]. Train detection (Train) is based on video of nearby train tracks that we have captured in our camera deployment, and have hand labeled. The remaining classifiers are trained on a video of a nearby intersection, also captured in our camera deployment. We have obtained the necessary permissions and plan to make our Trains and Intersection video dataset available publicly. We reserve a portion of these datasets to create synthetic video workloads for testing.

We use M-Trainer to produce a task-specific M-Package for each application. Model candidates are fine-tuned using the MobileNets-224 model pretrained on ImageNet as a base DNN (implemented in Keras [8] using TensorFlow [2]). Each M-Package contains several models with different degrees of fine-tuning as described in Section 5. We evaluate Mainstream using the M-Packages and hold-out validation sets from our datasets. Our experiments use the applications in Table 2.

Hardware. Training is performed on nodes equipped with Intel® Xeon® E5-2698Bv3 processors (2.0 GHz, 16 cores) and an Nvidia Titan X GPU. All end-to-end experiments use an Intel® NUC with an Intel® Core™ i7-6770HQ processor and 32 GiB DRAM, which is intended to represent an edge processing device. The Train and Intersection videos were captured using an Allied Vision Manta G-1236 GigE Vision camera.

8 Evaluation

We evaluate our system in the context of independent DNN-based video processing applications sharing a fixed-resource edge computer. In our evaluation, we show that Mainstream's dynamic approach outperforms static solutions in all of our experimental settings, across various application workloads, computational budgets, and numbers of concurrent applications. Mainstream's X-voting is capable of improving F1-score but, like model specialization, must also be dynamically configured to the resources available. In addition to our benchmarked applications, we show an end-to-end Mainstream deployment of a train detection application used for environmental pollution monitoring.

8.1 Mainstream Improves App Quality

Our goal in event detection is to maximize per-*event* F1-score. We compare the F1-score achieved by Mainstream with two baselines: No-Sharing and Max-Sharing. *No-Sharing* is the default behavior for a multi-tenant environment and is the approach taken by systems like TensorFlow Serving [1] and Clipper [12]. No-Sharing maximizes classification accuracy at the cost of a reduced sampling rate and requires no coordination between tenants. *Max-Sharing* is the sharing approach used by MCDNN [18]. It uses partial-DNN sharing by fine-tuning the final layer of concurrent DNNs. In many cases, Max-Sharing provides better F1-score relative to No-Sharing when a non-trivial number of applications share the infrastructure; it sacrifices classification accuracy to maximize the number of frames processed. We show, however, that Max-Sharing is less effective than making deliberate runtime decisions about how much sharing to use.

In order to observe the effects of increasingly constrained resources without a large number of distinct applications, we generate additional applications by augmenting our application set. Each of the seven classification tasks in Table 2 has a corresponding "accuracy-tradeoff curve", which represents the relationship between per-frame accuracy and the shared stem size (Fig. 4b). For each application in our experiments, we randomly choose one of the seven classifiers (and its corresponding accuracy-tradeoff curve) and parameterize it with a different event length, event frequency and inter-frame correlation. To capture the effects of diverse application characteristics, the parameters are uniformly sampled from a range of possible values. Each workload consists of up to 30 concurrent applications. In most experiments, we show the behavior averaged across 100 workloads. Our video capture rate for all experiments is 10 FPS.

Mainstream outperforms static approaches. M-Scheduler maximizes per-event F1-score by varying the sampling rate and amount of sharing. Each additional application introduces more resource contention, forcing the system to pick a different balance between accuracy and sampling rate to achieve the best average F1-score.

Detection Task	Dataset Description	Number of Images	Avg Event Length	Min Event Length	Event Frequency
Pedestrians	Urban Tracker atrium video	4538	59	2	0.63
Bus	Intersection near CMU video	4762	1039	141	0.27
Red Car	Intersection near CMU video	9172	228	46	0.08
Scramble	Intersection near CMU video	1500	412	382	0.16
Schoolbus	Intersection near CMU video	2600	854	92	0.03
Trains	Train tracks near CMU video	3066	132	20	0.01
Cars	Images of 23 car models	3042	—	—	—

Table 2: Labeled datasets used to train classifiers for event detection applications. Average and minimum event lengths are reported in number of frames. Event length and event frequency only apply to video datasets and not Cars.

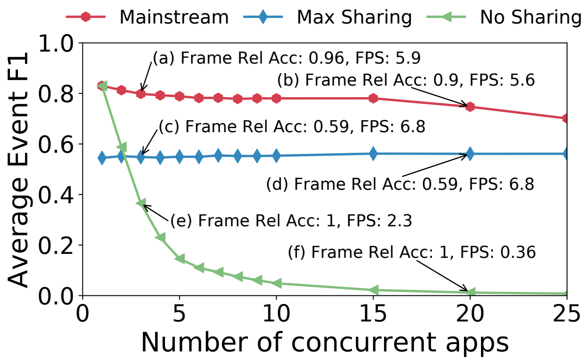


Figure 6: Mainstream improves F1-scores vs. No-Sharing between applications or conservatively sharing all layers but the last one. "Frame Rel Acc" is the relative image-level accuracy of the model deployed, compared to the best performing model candidate. "FPS" is the average observed throughput of the deployed applications.

Fig. 6 compares Mainstream with our baseline strategies. Mainstream delivers as much as a 87X higher per-event F1-score than No-Sharing and as much as a 47% higher score vs Max-Sharing. Fig. 6 reports F1-scores averaged across 100 workloads. The relationship between the three schedulers holds when examining individual workloads. No-Sharing exhibits low recall because of its low throughput—the system has fewer opportunities to detect the event. Max-Sharing has high throughput but a worse precision because the underlying model accuracy is lower—it evaluates many frames but does so inaccurately. Mainstream outperforms by striking a balance, sometimes choosing a more accurate model, and sometimes choosing to run at a higher throughput.

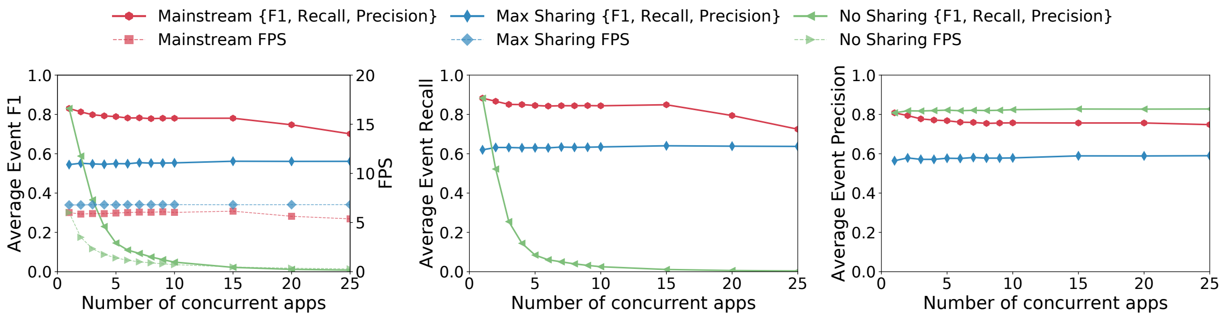
Mainstream dynamically balances precision and recall. Fig. 7 delves into the system effects of Mainstream more deeply. F1-score, recall, and precision are plotted. The average application frame rate is plotted, showing how Mainstream dynamically tailors resource usage to the workload. (Not shown is the varying model accuracy.) Optimizing for precision requires careful tuning of the application frame rate. While higher FPS always leads to higher recall, it does not always lead to higher precision. (A high frame rate may only increase false positives without increasing expected true positives.) For instance, No-Sharing's low frame rate and high per-frame accuracy allows it to have the highest precision of the three approaches. When given just a few applications, Mainstream runs specialized models, while throttling the stream rate to avoid unnecessary false positives. As resources become scarce, many applications begin to share more of the network.

Mainstream improves upon Max-Sharing even under tight resource constraints. Fig. 8 shows the effect of Mainstream, Max-Sharing, and No-Sharing on a range of computational budgets. We average the event F1-scores across 100 workloads, each with 3 applications. The right-most points represent the scenario of running on computational resources equivalent to an Intel$^{\circledR}$ NUC. With a small workload of three applications, No-Sharing performs better than Max-Sharing, as it is able to run expensive models at a high enough frame rate. As we decrease the available budget, Max-Sharing's conservative sharing approach allows it to be more scalable than No-Sharing. However, even after the computational budget is reduced by 83%, Mainstream still improves application performance, compared to the overly conservative Max-Sharing approach.

8.2 Tuned X-Voting improves F1-score

Applications that suffer from low per-frame precision will generate many false positives. An X-voting approach can greatly decrease the incidence of false positives, as X consecutive classifications are needed in order to report a detection. Too large a value of X can hurt recall, causing real events to go unreported. By using X-voting and optimizing the parameter X, Mainstream can improve the overall average event F1-score.

Fig. 9 shows the effects of X-voting on F1-scores as X and the number of applications are varied, while total resources are kept fixed. With just a few concurrent applications, running at high frame rates, 7-voting and 5-voting yield the highest F1-scores. With more resource



(a) F1-score

(b) Recall

(c) Precision

Figure 7: The average event F1-score (Fig. 7a), recall (Fig. 7b) and precision (Fig. 7c) across 100 deployed workloads are shown (solid lines) alongside the average frame-rate across applications (dotted lines). Mainstream dynamically balances recall and precision to maximize aggregate F1-score. With high numbers of concurrent applications, Mainstream sacrifices small amounts of both specialization and frame-rate.

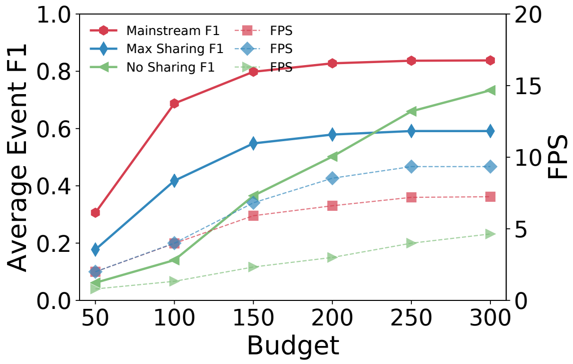


Figure 8: Mainstream improves F1-score of workloads under varying computational budgets. The rightmost points on the X axis represent the resources available on an Intel® NUC. Even under heavy resource constraints, there is available capacity for Mainstream to perform optimizations.

contention, and lower throughput, 3-voting becomes the best choice as the cost of dropping true positives outweighs the benefit of reducing false positives for higher values of X. When resources become too constrained, this approach is less viable, e.g., 1-voting becomes the best approach at 25 concurrent apps. Fig. 9 also shows the Pareto frontier of F1-scores achievable across all values of X for a given number of concurrent applications.

8.3 Mainstream Deployment

We deployed our environmental pollution monitor application and nine other concurrent applications using both Mainstream and a conventional No-Sharing approach for one week on the hardware setup described in Section 7. Fig. 10 shows the trace of both approaches on a *single* train event sequence, indicating the frames analyzed. A hit represents a correct classification of the train, a miss represents an incorrect classification. We see that Mainstream's deployment samples the stream more frequently, yielding many more hits (and misses) than No-Sharing;

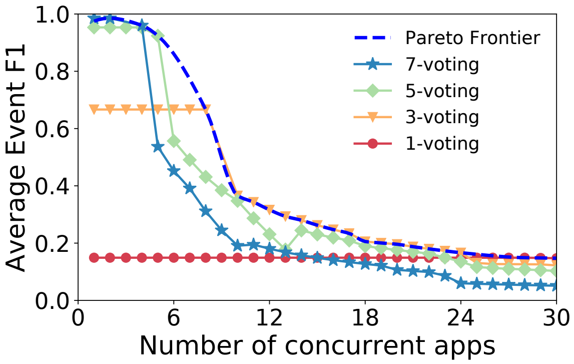


Figure 9: X-voting increases precision and helps Mainstream achieve a higher F1-score, but only if frame rate is high enough to avoid hurting recall. Thus, the effects vary by the level of resource contention. The Pareto frontier shows the F1-scores achievable given the dynamic selection of an optimal X-voting scheme for the resource scenario.

the result, though, is that Mainstream detects the train event earlier and more confidently.

We control the false positive rate with 2-voting, requiring Mainstream to have two positive samples before an event is classified. The false positive rate of the Train video drops from 0.028 to 0.00056. No-Sharing and Mainstream achieve a 0 and a 0.00056 false positive rate, respectively. In the analyzed deployment in Fig 10, we see that Mainstream still detects the train easily and quickly.

9 Additional Related Work

Several recent systems have attempted to tackle the problem of optimizing execution of visual computing pipelines.

VideoStorm [48] is a video analytics system for large-scale clusters and workloads. It analyzes resource use and application-goal-based metrics as a function of tunable parameters of the analytics pipelines, building models for each application independently. It uses these models

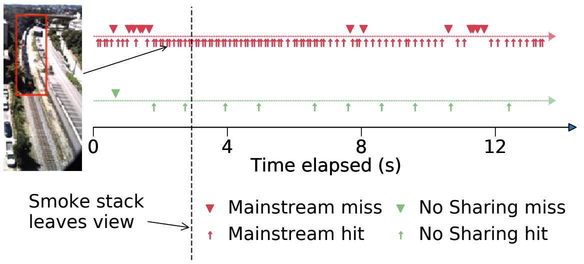


Figure 10: Timeline of Mainstream running a train detector app with 9 concurrent applications. Our goal is to detect the train as early as possible, before the smoke stack is out of view (end of window represented by the dotted line). Mainstream detects the train earlier and more confidently than No-Sharing.

to allocate resources and select parameters for deployed applications on a target platform, in order to maximize application quality metrics. VideoStorm takes a black-box view of the applications, and assumes that quality and resource consumption of co-deployed applications are independently determined. Therefore, it cannot take into account computation sharing, or optimize the sharing vs. degree of specialization tradeoff. In contrast, Mainstream takes a white-box approach to modeling application quality, and can explicitly tune computation sharing to improve application quality metrics. Compared to VideoStorm, Mainstream sacrifices some generality to solve the joint optimization problem.

MCDNN [18] introduces a static approach to sharing DNN computation, in which each application developer independently determines their amount of model specialization. MCDNN opportunistically shares any identical unspecialized layers between applications. In contrast, Mainstream's training and scheduling components allow late binding and jointly-optimized selection of the degrees of specialization at run time, when resource availability and co-deployed tasks are known.

Inference serving systems. Mainstream is an inference serving system for running neural networks on resource-constrained nodes. Other inference serving systems include Clipper [12], NoScope [26], and TensorFlow Serving [1]. Like Mainstream, these systems optimize for latency and throughput gains. Clipper caches results from multiple models, dynamically chooses from the results, and optimizes the batch size. NoScope replaces expensive neural networks for object detection with cheaper difference detectors and specialized models. TensorFlow Serving increases throughput with batching and hardware acceleration. LASER [4] and Velox [11] are inference serving systems for non-DNN models. LASER deploys linear models while Velox deploys personalized prediction algorithms using Apache Spark.

Unlike Mainstream, these inference serving systems do not share computation between independently trained models. They also target cluster environments. Mainstream targets edge devices with limited resources, where achieving the right degree of DNN computation sharing is particularly important, though such sharing would also be valuable in large data centers.

Reducing DNN inference time. Approaches to reducing DNN inference time for vision applications can be broadly classified into those that reduce model precision [16, 50, 10, 9, 7, 23, 39], use efficient network architectures [20, 24], use anytime prediction methods [22, 21], or employ model compression and sparsification [17, 33, 46]. All of these methods are orthogonal to Mainstream's adaptive DNN computation sharing technique, but share its goal of selecting the right trade-off between per-frame quality and frame throughput.

Multi-task networks. Multi-task learning [6, 49, 5, 36, 35, 34, 27, 38] is a ML approach in which a single model is trained to perform multiple tasks. Using multiple tasks to train a single model helps achieve better accuracy because of better generalization and complementary information [6, 41]. In the context of DNNs, a multi-task network can have a varying number of shared layers across tasks and task-specific layers [36, 35]. Multi-task learning assumes that all of the tasks are known *a priori*, and that training data for all of the tasks is available for use in a single training process. In contrast, Mainstream allows each task to be developed, trained, and deployed independently, and avoids the need to share or expose proprietary or privacy-sensitive training data between task developers. Note that one can run a multi-task network as a single large application in Mainstream.

10 Conclusion

Mainstream adaptively orchestrates DNN stem-sharing among concurrent video processing applications sharing the limited resources on an edge device, resulting in much higher aggregate application quality. Experiments with several event detection tasks confirm that Mainstream significantly increases overall event F1-score relative to current approaches over a range of concurrency levels.

11 Acknowledgments

We thank the member companies of the PDL Consortium (Alibaba, Broadcom, Dell EMC, Facebook, Google, HPE, Hitachi, IBM Research, Intel, Microsoft, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Toshiba, Two Sigma, Veritas and Western Digital) for their interest, insights, feedback, and support. This work is supported in part by funding from Intel as part of the Intel STC for Visual Cloud Systems (ISTC-VCS). We also thank Michael Sussman for help in developing statistical models.

References

[1] Tensorflow Serving. https://www.tensorflow.org/serving/.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[3] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan. YouTube-8M: A large-scale video classification benchmark. *CoRR*, abs/1609.08675, 2016. URL http://arxiv.org/abs/1609.08675.

[4] D. Agarwal, B. Long, J. Traupman, D. Xin, and L. Zhang. Laser: A scalable response prediction platform for online advertising. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14. ACM, 2014. URL http://doi.acm.org/10.1145/2556195.2556252.

[5] K. Ahmed and L. Torresani. Branchconnect: Large-scale visual recognition with learned branch connections. *CoRR*, abs/1704.06010, 2017. URL http://arxiv.org/abs/1704.06010.

[6] R. Caruna. Multitask learning. In *Learning to learn*, pages 95–133. Springer, 1998.

[7] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15. JMLR.org, 2015. URL http://dl.acm.org/citation.cfm?id=3045118.3045361.

[8] F. Chollet et al. Keras. https://keras.io, 2015.

[9] M. Courbariaux and Y. Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *Advances in Neural Information Processing Systems*, 2016.

[10] M. Courbariaux, Y. Bengio, and J. David. Binaryconnect: Training deep neural networks with binary weights during propagations. 2015.

[11] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. In *CIDR*, 2015.

[12] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA, 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[14] D. Eigen, C. Puhrsch, and R. Fergus. Depth map prediction from a single image using a multi-scale deep network. In *Advances in neural information processing systems*, pages 2366–2374, 2014.

[15] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Region-based convolutional networks for accurate object detection and segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 38(1):142–158, 2016.

[16] Y. Gong, L. Liu, M. Yang, and L. D. Bourdev. Compressing deep convolutional networks using vector quantization. *CoRR*, abs/1412.6115, 2014. URL http://arxiv.org/abs/1412.6115.

[17] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. *SIGARCH Comput. Archit. News*, 44(3):243–254, June 2016. ISSN 0163-5964. doi: 10.1145/3007787.3001163. URL http://doi.acm.org/10.1145/3007787.3001163.

[18] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *Conference MobiSys'16 The 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobieSys '16. ACM, 2016.

[19] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[20] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. URL http://arxiv.org/abs/1704.04861.

[21] H. Hu, D. Dey, J. A. Bagnell, and M. Hebert. Anytime neural networks via joint optimization of auxiliary losses. *arXiv preprint arXiv:1708.06832*, 2017.

[22] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger. Multi-scale dense convolutional networks for efficient prediction. *arXiv preprint arXiv:1703.09844*, 2017.

[23] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *CoRR*, 2016. URL http://arxiv.org/abs/1609.07061.

[24] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. 2016. URL http://arxiv.org/abs/1602.07360.

[25] Jodoin, J.-P., Bilodeau, G.-A., and N. Saunier. Urban tracker: Multiple object tracking in urban mixed traffic.

In *IEEE Winter conference on Applications of Computer Vision (WACV14)*, March 2014.

[26] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: Optimizing neural network queries over video at scale. In *Proceedings of the VLDB Endowment, Vol. 10, No. 11*, 2017.

[27] I. Kokkinos. Ubernet: Training a 'universal' convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory. *CoRR*, abs/1609.02132, 2016. URL http://arxiv.org/abs/1609.02132.

[28] J. Krause, M. Stark, J. Deng, and L. Fei-Fei. 3d object representations for fine-grained categorization. In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia, 2013.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[30] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[31] T. J. Lee, J. Gottschlich, N. Tatbul, E. Metcalf, and S. Zdonik. Precision and recall for range-based anomaly detection. *SysML*, Feb 2018.

[32] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: common objects in context. pages 740–755", 2014.

[33] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Penksy. Sparse convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[34] M. Long and J. Wang. Learning multiple tasks with deep relationship networks. *CoRR*, abs/1506.02117, 2015. URL http://arxiv.org/abs/1506.02117.

[35] Y. Lu, A. Kumar, S. Zhai, Y. Cheng, T. Javidi, and R. S. Feris. Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification. *CVPR*, 2016. URL http://arxiv.org/abs/1611.05377.

[36] I. Misra, A. Shrivastava, A. Gupta, and M. Hebert. Cross-stitch Networks for Multi-task Learning. In *CVPR*, 2016.

[37] M. Oquab, L. Bottou, I. Laptev, and J. Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[38] R. Ranjan, V. M. Patel, and R. Chellappa. Hyperface: A deep multi-task learning framework for face detection, landmark localization, pose estimation, and gender recognition. *arXiv preprint arXiv:1603.01249*, 2016.

[39] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[40] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. CNN features off-the-shelf: An astounding baseline for recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR Workshops 2014, Columbus, OH, USA, June 23-28, 2014*, pages 512–519, 2014. doi: 10.1109/CVPRW.2014.131. URL http://dx.doi.org/10.1109/CVPRW.2014.131.

[41] S. Ruder. An overview of multi-task learning in deep neural networks. abs/1706.05098, 2017. URL http://arxiv.org/abs/1706.05098.

[42] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[43] K. Simonyan and A. Zisserman. Two-stream convolutional networks for action recognition in videos. In *Advances in neural information processing systems*, pages 568–576, 2014.

[44] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.

[45] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *CVPR*, pages 1701–1708, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-5118-5. doi: 10.1109/CVPR.2014.220. URL http://dx.doi.org/10.1109/CVPR.2014.220.

[46] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. *Advances in Neural Information Processing Systems*, 2016. URL http://arxiv.org/abs/1608.03665.

[47] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3320–3328, Cambridge, MA, USA, 2014. MIT Press. URL http://dl.acm.org/citation.cfm?id=2969033.2969197.

[48] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, NSDI '17, 2017.

[49] Z. Zhang, P. Luo, C. C. Loy, and X. Tang. Facial landmark detection by deep multi-task learning. In *ECCV*, 2014.

[50] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[51] S. Zilberstein. Using anytime algorithms in intelligent systems. In *AI Magazine, 17(3):73-83*, 1996.

VIDEOCHEF: Efficient Approximation for Streaming Video Processing Pipelines

Ran Xu$^{\alpha}$, Jinkyu Koo$^{\alpha}$, Rakesh Kumar$^{\alpha}$, Peter Bai$^{\alpha}$, Subrata Mitra$^{\beta}$
Sasa Misailovic$^{\gamma}$, Saurabh Bagchi$^{\alpha}$
α: Purdue University, β: Adobe Research, γ: University of Illinois

Abstract

Many video streaming applications require low-latency processing on resource-constrained devices. To meet the latency and resource constraints, developers must often approximate filter computations. A key challenge to successfully tuning approximations is finding the optimal configuration, which may change across and within the input videos because it is content-dependent. Searching through the entire search space for every frame in the video stream is infeasible, while tuning the pipeline offline, on a set of training videos, yields suboptimal results.

We present VIDEOCHEF, a system for approximate optimization of video pipelines. VIDEOCHEF finds the optimal configurations of approximate filters at runtime, by leveraging the previously proposed concept of canary inputs—using small inputs to tune the accuracy of the computations and transferring the approximate configurations to full inputs. VIDEOCHEF is the first system to show that canary inputs can be used for complex streaming applications. The two key innovations of VIDEOCHEF are (1) an accurate error mapping from the approximate processing with downsampled inputs to that with full inputs and (2) a directed search that balances the cost of each search step with the estimated reduction in the run time.

We evaluate our approach on 106 videos obtained from YouTube, on a set of 9 video processing pipelines with a total of 10 distinct filters. Our results show significant performance improvement over the baseline and the previous approach that uses canary inputs. We also perform a user study that shows that the videos produced by VIDEOCHEF are often acceptable to human subjects.

1 Introduction

Video processing has brought many emerging applications such as augmented reality, virtual reality, and motion tracking. These applications implement complex video pipelines for video editing, scene understanding, object recognition and object classification [14, 49]. They often consume significant computational resources, but also require short response time and low energy consumption. Often, the applications need to run on the local machines instead of the cloud, due to latency [14], bandwidth [50], or privacy constraints [46].

To enable low-latency and low-energy video processing, we leverage the the fact that most stages in the video pipeline are inherently *approximate* because human perception is tolerant to modest differences in images and many end goals of video processing require only estimates (*e.g.*, detecting object movement or counting the number of objects in a scene [5]). Many *domain-specific* algorithms have exposed algorithmic knobs, that can e.g., subsample the input images or replace expensive computations with lower-accuracy but faster alternatives [45, 41, 13, 26]. To complement domain-specific approximations, researchers have proposed various generic *system-level techniques* that expose additional knobs for optimizing performance and energy of applications while trading-off accuracy of the results. The techniques span compilers [39, 30, 42, 7, 36, 27], systems [3, 15, 18, 17, 31], and architectures [32, 29, 37, 36, 8].

Content-dependent Approximation. A fundamental challenge of uncovering the full power of both generic and domain specific approximations is finding the configurations of these approximations that provide maximum savings, while providing acceptable results for *each given input*. This challenge has two main parts.

First, the optimal approximation setting is dependent on the *content of the video*, not just on the algorithms being used in the processing pipeline. Often individual videos, or parts of the same video should have different approximation settings, requiring the program to make the decisions at runtime. Second, the optimization needs to explore a large number of approximate configurations before selecting the optimal one for the given input, requiring the optimizer to construct off-line models. Systems like Green [3] and Paraprox [36] dynamically adapt the computation using runtime checks of the intermediate results, while Capri [43] selects approximation level from the input features at the program start. However, the systems rely on extensive off-line training to map the approximation levels to accuracy and performance.

To relax the dependency on off-line training, Laurenzano *et al.* [25] propose Input Responsive Approximation (IRA) for runtime recalibration with no offline training. IRA creates *canary inputs*, smaller representations of the full inputs (obtained via subsampling), and then reruns the computation on the canary input with different approximation settings, until it finds the most efficient setting that maintains the accuracy requirement (on the canary). While the concept is promising, the application of IRA to video processing pipelines is limited:

- IRA has been applied to individual computational kernels (in contrast to full pipelines). It is unclear how to capture the interactions between the stages of the pipeline, how often to calibrate, and what are the optimal canary sizes.
- IRA uses the approximation settings derived from the canary input to the full input, assuming that the errors for the full and correlated inputs will be *identical*. However, this assumption is often incorrect (98% of cases, Figure 2) and leads to missed speedup opportunities.
- IRA's greedy search may introduce additional overheads and may not find good approximation settings efficiently because it has no notion of what are the appropriate points in the stream to search.

Our Solution: VIDEOCHEF. We present VIDEOCHEF, a fast and efficient processing pipeline for streaming videos. VIDEOCHEF can optimize the performance subject to accuracy constraints for the system-level and domain-specific approximations of all kernels in the video processing pipeline. Figure 1 presents VIDEOCHEF's end-to-end workflow:

- Like IRA, VIDEOCHEF uses small-sized canary input to guide the on-line search for approximation setting. However, unlike IRA, VIDEOCHEF is tailored for optimization of the whole video processing pipelines, not just individual kernels.
- In contrast to IRA, VIDEOCHEF presents a finely tunable prediction model for mapping the error from the canary input to that with the original input. This prediction model is trained offline and hence does not generate any additional runtime overhead. At the same time, it is much more lightweight than the full off-line training employed by other approaches.
- At runtime, VIDEOCHEF performs an efficient search through the space of approximation settings and ensures that the cost of the search does *not* overwhelm the benefit of approximating the computation.

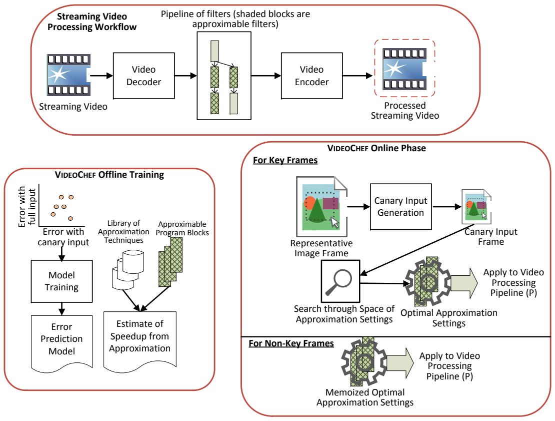

Streaming Video Processing Workflow
Pipeline of filters (shaded blocks are approximable filters)
Streaming Video
Video Decoder
Video Encoder
Processed Streaming Video
VIDEOCHEF Offline Training
Error with full input
Error with canary input
Library of Approximation Techniques
Approximable Program Blocks
Model Training
Error Prediction Model
Estimate of Speedup from Approximation
VIDEOCHEF Online Phase
For Key Frames
Representative Image Frame
Canary Input Generation
Canary Input Frame
Search through Space of Approximation Settings
Optimal Approximation Settings
Apply to Video Processing Pipeline (P)
For Non-Key Frames
Memoized Optimal Approximation Settings
Apply to Video Processing Pipeline (P)

***Figure 1:** End-to-end flow of approximate video processing with* **VIDEOCHEF.** *The video processing pipeline comprises multiple filters, which can be approximated to save computation at the expense of tolerable video quality. The offline and the online components of* **VIDEOCHEF** *work together to determine the best approximation setting for each approximable filter block.*

We evaluate VIDEOCHEF with three error models and two search strategies, applied to a corpus of 106 YouTube videos from 8 content categories, which span the range of video features (e.g., color and motion). We analyze 10 filters arranged in 9 pipelines of size 3. We find that VIDEOCHEF is able to reach within 20% of the theoretical best performance possible and outperforms IRA's performance by 14.6% averaged across all videos and saves on an average 39.1% over the exact computation given a relatively restrict quality requirement. While given a more loose quality requirement, VIDEOCHEF is able to reach within 26.6% of the theoretical best and also achieve higher performance gain – 53.4% and 61.5% over IRA and exact computation, respectively.

While we have framed this discussion in terms of video processing, the novel contributions outlined below apply to other low-latency streaming applications, with the fundamental requirement that the characteristics change to some extent from one segment of the stream to another, for instance, online video gaming, augmented reality and virtual reality applications.

Contributions. We make the following contributions:

1. We present VIDEOCHEF, a system for performance and accuracy optimization of video streaming pipelines. It consists of off-line and on-line components, that together adapt the application's approximation level to the desired output quality.
2. We build a predictive model to accurately estimate the quality degradation in the full output from the error generated when using the canary input. This enables more aggressive approximation setting the approximation algorithm that has tunable knobs.
3. We propose an efficient and incremental search technique for the optimal approximation setting that

takes hints from the video encoding parameters to reduce the overhead of the search process.

4. We demonstrate the benefits of VIDEOCHEF through (1) quantitative evaluation on various real-world video contents and filters and (2) a user study.

2 Background and Motivation

Error Metric: At a high-level, a video is composed of a sequence of image frames. To quantify the error in the output or the processed video due to approximation, we measure the Peak Signal-to-Noise Ratio (PSNR) of the output video. PSNR is the average of the PSNRs of the individual frames in the output video. Suppose that a video consists of K frames where each frame has $M \times N$ pixels. Let $Y_k(i,j)$ be the value of the pixel at (i,j) position on the k-th frame of the processed video without any use of approximation, and $Z_k(i,j)$ be the value of the pixel when approximation was applied. Then, the PSNR of the approximate output is computed as follows:

$$PSNR = \frac{1}{K}\sum_{k=0}^{K-1} 20 \times log_{10}\frac{MaxValue}{\sqrt{MSE(Z_k,Y_k)}}, \qquad (1)$$

where $MaxValue$ is the maximum possible pixel value present in the frame, and $MSE(Z_k,Y_k)$ is the mean square error between Z_k and Y_k, i.e., $\sum_i\sum_j(Z_k(i,j)-Y_k(i,j))^2$, as a result of approximation. Thus, lower the PSNR, the higher the error in the output video.

Isn't the problem solved by IRA and Capri? IRA (Input Responsive Approximation) [25] and Capri [43] attempted to address the problem of selecting optimal approximation level for individual inputs.

IRA [25] solely relies on canary inputs to search for best approximation settings. Thus, it implicitly assumes that the magnitude of error corresponding to a particular approximation setting on the *canary inputs* is identical to the error with the same approximation settings on the *full-sized inputs*. But, Figure 2 shows our experiment with 424 real images and 216 different approximation settings. We found that for the same approximation settings, the PSNR of the full-sized inputs can be significantly different from the PSNR of the canary inputs. Most of the points (about 98%) are above the diagonal, indicating that the error on the full input is lower than that with the canary input for the same approximation level.

We attribute the difference in the approximation to the higher variations between neighboring pixel values for canary inputs. Therefore, for the same approximation settings, the approximate processing on canary inputs gives lower PSNR. We found that on an average, the PSNR of a full-sized output is 5.36 dB higher than the PSNR of canary output. Therefore, IRA misses an opportunity for more aggressive optimization that can fit within the user-specified quality threshold.

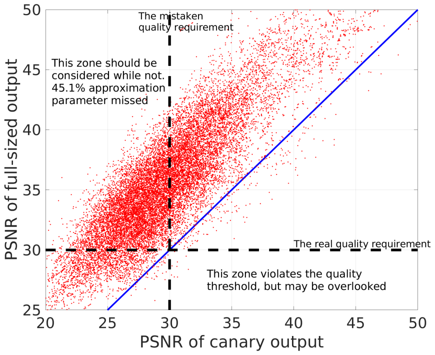


Figure 2: The PSNR of full-sized output versus the PSNR of canary output, for the I-frames of 106 videos on one of our application Boxblur-Vignette-Dilation video filter pipeline. The PSNR of full output is higher for over 98% approximation settings and 45.1% of the approximation settings lie in such a zone that is ignored by IRA approach but actually satisfies the quality requirement.

Capri [43] rigorously addresses the problem of selecting the best approximation settings to minimize the computational cost, while meeting the error bound. But Capri also fails in the video processing setting because it does not recalibrate itself with the stream and thus cannot change its approximation settings when the characteristics of the stream change. Further, it (1) relies on prior enumeration of all possible inputs, which is impossible in this target domain, and (2) performs the selection of approximation settings completely offline, which reduces the cost of the optimization but makes it non-responsive to changes in the input data.

3 Solution Overview

Figure 1 shows the end-to-end workflow of streaming video processing, with approximation.

Approximation. Under normal processing, a video decoder converts the video into its constituent frames. Then a sequence of "filters" (synonymously, processing steps or pipeline stages) is applied to each frame. Examples include blurring filter (*e.g.*, at the TSA airport checkpoint scanners) and edge detection filter (*e.g.*, for counting people in a scene). Finally, the transformed frames are optionally put together by a video encoder. To make such processing fast and resource efficient, VIDEOCHEF intelligently uses selective approximation (Section 3.1) during the computation of the filters. The user sets the quality constraint on the output video quality. An example specification is that the PSNR of the output video should be above 30 dB.

Accuracy Calibration with Canary Inputs. For each representative frame (called "key frame" here), VIDEOCHEF determines a *canary input* (Section 3.2), which summarizes the full frame such that the dissimilarity between the full and the canary frame remains below a threshold. With the canary input, VIDEOCHEF occasionally recalibrates the approximation levels of the filters. It determines when to call the search algorithm using domain specific knowledge about the frames and scenes (Section 3.3). For this, we extract hints from the video decoder which lets VIDEOCHEF determine the key frames. This amortizes the cost of the search across many frames of the video, with 80-120 frames being a typical range for MPEG-4 videos. In the absence of such a video decoder, we have a variant, which triggers the search upon a scene change detection.

Online Search for Optimal Tradeoffs. VIDEOCHEF searches for the approximation setting of each filter that gives the lowest execution time subject to a threshold for the output quality (Section 3.4). Since the search for approximation is done with the *canary input*, the error of approximate computation is different from the error of the computation on the full input. VIDEOCHEF introduces a method to accurately map between these two errors (Section 3.5). In performing this estimation, we consider multiple variants of VIDEOCHEF, depending on what features are available to the predictor, such as, some categorization of the video frame according to its image properties. Through this procedure, we aim to maximally leverage the approximation potential in the application and give flexible approximation choices.

3.1 Approximation techniques

The computations involved in filtering operations can be approximated by VIDEOCHEF in various ways as long as each of the underlying approximation techniques exposes knobs that can be tuned to control the approximation levels (ALs). For example, in the three popular program transformation-based approximation techniques, the variable `approx_level` is a tuning knob that controls the levels of approximation. A higher value implies more aggressive approximation, leading most often to higher speedup but also higher error. These transformations are performed automatically by a compiler (LLVM in our case).

Loop perforation: In loop perforation [42, 30], the computation is reduced by skipping some iterations, as shown below.

```
for (i = 0; i < n; i = i + approx_level)
  result = compute_result();
```

Loop truncation: In loop truncation [42, 30], the last few iterations of the computation are dropped as shown in the following example:

```
for (i = 0; i < (n - approx_level); i++)
  result = compute_result();
```

Loop memoization: In this technique [7, 36], for some iterations in a loop we compute the result and cache it. For other iterations we use the previously cached results.

```
for (i = 0; i < n; i++)
  if (i % approx_level == 0)
     cached_result = result = compute_result();
  else result = cached_result;
```

3.2 Canary Inputs

To reduce the search overhead for finding the best approximation level within each frame of the video, we generate canary inputs for the frame following the work in [25]. A good canary input should meet two requirements: (1) it should be *close enough* to the original input so that the AL found by the canary is the same as the AL computed from the original; (2) it should be small enough that the search process using the canary input is efficient. We first define the dissimilarity metric to compare the canary sample video and the full-sized video and then show how to choose the appropriate canary input.

Metrics of Dissimilarity. We define two metrics of dissimilarity. Let a full-sized video have K frames and $M \times N$ pixels in each frame, and each pixel has the property $X(i,j)$. A canary video has K frames with $m \times n$ pixels, and the same property $Y(i,j)$. The property could be one component in the YUV colorspace of an image, where the Y component determines the brightness of the color (known as "luminance") while the U and V components determine the color itself (known as "chroma") and each ranges from 0 to 255. The "dissimilarity metric for mean" (SMM), is defined as follows (following [25]):

$$mFull = \frac{1}{M \times N \times K} \sum_{i=0}^{K-1} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} X(i,j) \tag{2}$$

$$mSmall = \frac{1}{m \times n \times K} \sum_{i=0}^{K-1} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} Y(i,j) \tag{3}$$

$$SMM = \frac{|mSmall - mFull|}{mFull} \tag{4}$$

When a pixel has a vector of values, such as the YUV colorspace which has 3 values for the 3 components, then the SMM metric is combined across the different elements of the vector. The combination could be a simple average or a weighted average; we use the latter due to the higher weight of the Y-channel in the YUV colorspace. Similarly, we define the "dissimilarity metric of standard deviation" (SMSD) to capture the dissimilarity in the Standard Deviation between the full input and the canary input.

Generating Candidate Canary Videos. Given a frame of the video from which to generate the canary video, we resize the frame to a fraction $1/N$ of its original size to create the canary video. Typical sizes that we find useful in our target domain are $1/16$, $1/32$, $1/64$, $1/128$, $1/256$ of the original size. Since the frame is a 2-D matrix of pixels, to resize it to $1/N$ of its original size, we shrink the width and height each to $1/\sqrt{N}$ of the full size by sub-sampling 1 pixel out of every $\sqrt{N}$ pixels.

Reducing an input size causes at least proportional reduction in the amount of work inside the filter. Many filters that are finding increasing use are super-linear, where the benefit of using a small canary frame is even more significant. Two popular examples are determining optical flow to measure motion [4] and morphological filter [10], where the value of each pixel in the output image is based on a comparison of the corresponding pixel in the input image with its neighbors. We compute the similarity between the full-sized video frame and the canary video frame according to the metrics SMM and SMSD. We set the maximum dissimilarity metric we can tolerate as a threshold parameter—we find 10% is a practically useful threshold for both SMM and SMSD. Among all the qualified canary inputs, we select the smallest one as our final choice.

3.3 Identifying Key Video Frames

Searching for the best AL for each approximable program block is computationally expensive. Conceptually, we would want to repeat the search when the characteristic of the video changes significantly so that the optimal approximation setting is expected to be different. In practical terms, we want to perform such change point detection without having to parse the content of the video. Video encoders already provide hints when the content of the scene has changed significantly.

We make the observation that videos have temporal locality, and many frames in the *same group* will have the same approximation setting. Therefore, we can perform a single search once per a single group of frames. We leverage domain-specific knowledge about videos to automatically select the group boundaries in two ways:

Scene Change Detector. Our first observation is to recalibrate the approximation at the beginning of different scenes. This approach is general and works for any video format. There are mainly 2 classes of scene change detectors, namely, pixel-based and histogram-based. The pixel-based methods are highly sensitive to camera and object motion. Histogram-based methods are good for detecting abrupt scene changes. To keep our overhead low (since the detection algorithm runs on every frame), we limit ourselves to detecting only abrupt scene changes and use canary frames for this detection.

We implement a histogram-based scene change detector using only the Y-channel of the canary frames [20]. We experimentally found the Y-channel information was sufficient to detect abrupt scene changes and we were more concerned about overhead of scene change detector than its accuracy. The algorithm detects a scene change whenever the sum of the absolute difference across all the bins of histograms of two consecutive frames is greater than some predefined threshold (20% of the total pixels in our evaluation). Our experiments show that the optimal configuration (found through offline brute-force search) changes dramatically at scene change boundaries but stay relatively stable within a group of frames.

I-frame Selection for MPEG videos. The second solution takes advantage of I-frames, present in the popular H.264 encoder (which the MPEG-4 and many other video formats follow). It defines three main types of frames: $I-$, $P-$, and $B-frames$ [21]. An I-frame uses intra-prediction meaning the predicted pixels within this frame are formed using only samples from the same frame. The P- and the B-frames use inter-prediction meaning the predicted pixel within this frame depends on samples from the same frame *as well as* samples from other frames around it (the distinction between P- and B-frames is not relevant for our discussion).

When to insert an I-frame (also called a "reference frame") depends on the exact coding scheme being used, but in all such coding schemes that we are aware of, a big difference in the frame triggers the insertion of a new I-frame, since inter-coding will give almost as long a code as intra-coding. Further, because an I-frame does not have dependencies on other frames, this makes it easier to reconstruct and perform the (exact or approximate) computation. We see empirically that for a wide range of videos used in our evaluation, the average spacing between adjacent I-frames is 137 frames. Although specific to only some video formats, it results in a low sampling rate and consequently the low search overhead, while triggering search at a suitable granularity.

3.4 Search with Canary Inputs

An approximable program block exposes one or more approximation knobs. The `approx_level` variable mentioned with the loop-based approximation techniques in Section 3.1 is an example of such a knob. In our notation, we use AL 1 to denote the exact computation. The higher the AL is, the less accurate the computation is and the higher the speedup is. Now for a pipeline of cascaded filters, each having one or more approximation knobs we have a *vector* of approximation settings per frame. We define a *setting* in the processing pipeline as the combination of ALs for each of the approximable program blocks in the video processing pipeline. For example, with an *n*-stage processing pipeline and each stage being approximable and having exactly one approximation knob, the setting will be $\vec{A} = \{a_1, a_2, \cdots, a_n\}$, where a_i denotes the AL of knob i.

To find the best approximation setting, we follow a searching algorithm outlined as follows:

1. **Start searching** at a particular setting, typically $(1, 1, \cdots, 1)$, corresponding to no approximation.
2. **Select a group of candidate settings** by the Candidate Selection Algorithm. The selection algorithm simply selects the next set of settings to try out in the search process. The greedy algorithm works as follows: given the current setting $\vec{A^{(0)}}$, we assume that we can reach the best AL by looking at each step 1 AL further in each approximable block. So the candidates

are $\{\vec{A^{(i)}}\}$, with $i = 1 \cdots n$, for n approximable blocks and $\vec{A^{(j)}} = \{a_1^{(0)}, \cdots a_{j-1}^{(0)}, a_j^{(0)} + 1, a_{j+1}^{(0)}, \cdots, a_n^{(0)}\}$.

3. **Decide whether to continue search**, i.e., whether it is worthwhile to try any of these candidate settings. We use the Approximation Payoff Estimation Algorithm (Section 3.4.1). If not, return the current setting. This algorithm estimates whether the saving due to the more aggressive approximation can compensate for the time of the additional search step.
4. **Try each worthwhile candidate setting** from the set computed by the previous step. Use the ALs in candidate settings to run approximate computation on canary video and compute the error metric for each candidate setting. Then map the error metric to that with the full sized outputs.
5. **Check for exit or iterate** – if error metrics of all the full video outputs exceed the error boundary, return the current setting. Otherwise, the candidate setting which gives the lowest error becomes the next setting, go to (2) and iterate.

3.4.1 Approximation Payoff Estimation Algorithm

The goal of this algorithm is to estimate the benefit of executing the application with the new AL searched for versus the cost of searching with the new AL, all for the key frame under question. Let the current setting be represented by $\vec{A^{(0)}} = \{a_1^{(0)}, a_2^{(0)}, \cdots, a_n^{(0)}\}$. Recollect that this is the set of ALs for each of the approximable blocks in the application. Let the execution time of the application at setting $\vec{A^{(i)}}$ and with canary downsampling C_d be given by $g(A^{(i)}, C_d)$, where $C_d = 1$ denotes the execution time with the full input.

This algorithm works in a breadth-first fashion and attempts to prune some of the paths where exploring higher degrees of approximation for a particular knob cannot speedup the execution further and may lead to slowdown due to associated overheads. From the current setting of $\vec{A^{(0)}}$, let the next possible settings of exploration be $\vec{A^{(1)}}, \vec{A^{(2)}}, \cdots, \vec{A^{(N)}}$. For example, with greedy search, with n approximable blocks, there will be n possible next settings. The maximum possible benefit by exploring all the candidate next settings is calculated as:

$$B = \max_{i=1}^{N}[g(A^{(0)}, 1) - g(A^{(i)}, 1)] \tag{5}$$

This benefit B simply means the maximum reduction in execution time across all the possible candidate settings, when run with the full input. However, to realize this gain, we have to pay the cost of searching, which can be expressed in terms of the overhead as follows:

$$O = \sum_{i=1}^{N} g(A^{(i)}, C_d) \tag{6}$$

This overhead O is simply the cost of executing the application with the next step ALs, but with the canary input (and hence the downsampling ratio C_d). The decision for VIDEOCHEF becomes simple: if $B > O$, then continue the search, else stop and return the current setting.

3.5 Error mapping model

We have to develop an error mapping model to characterize the relation between error in the canary output and error in the full output, for the same approximation levels. This is important because we have seen empirically (Figure 2) that the canary errors are higher than full frame errors for most points. We propose three different mapping models to use according to different amounts of knowledge in the model.

3.5.1 Model-C

Suppose we know the error metric of a canary output C. The error metric of a full-sized output F is estimated by a quadratic regression model as follows,

$$F = w_0 + w_1 \times C + w_2 \times C^2 \tag{7}$$

Offline, we calculate the ground truth of the pairs (C, F) for every possible AL $\vec{A}$ and for all the videos in the training set. In practice, we find that sub-sampling the space of possible ALs still provides accurate enough training, with a sub-sampling rate of 10% being adequate. Let us say that the error bound specified by the user is E_B. Then clearly we want $F < E_B$. However, due to the possible inaccuracy of the error mapping model, we want to explore a larger space so that we are not missing out on opportunities for approximating. Therefore, while training the model, Model-C, we explore all the points where $F \le E_B + \Delta$, where Δ is a user configurable parameter for how far outside the tolerable region we want to explore in the model. Then we solve the unknown coefficients w_0, w_1, and w_2 in the model. We find empirically that for a large set of videos, this model reaches its limits with the quadratic regression function.

3.5.2 Model-CA

Now, suppose VIDEOCHEF has additional knowledge of what ALs were in effect. Given the error metric of a canary output C, which is computed approximately with ALs $\vec{A} = \{a_1, a_2, \cdots, a_n\}$, we construct the input vector $\vec{I} = (1, C, \vec{A})$. The first element of this vector is the constant 1 and allows for a constant term in our equation for F. Then the error metric of a full-sized output F is estimated by a regression model as follows,

$$F = \vec{I} \cdot w \tag{8}$$

where w is a $(n+2) \times 1$ coefficient vector. The goal of the training is to estimate the matrix w. The elements of the matrix w provide the weights to multiply the different input components—the error in canary output (C), and the different ALs. We use similar training method offline as for Model-C.

3.5.3 Model-CAD

Many of approximation techniques on image processing reduce computation load by skipping a fraction of rows of images. Thus, the difference over rows is often related with approximation quality. Inspired by this characteristic, we consider a new feature vector $\vec{D} = (d_1, d_2, d_3)$, where each of d_k's represents a feature extracted from one of Y, U, and V channels of an image. The feature d_k is referred to as a row-difference feature and is defined as the mean of absolute difference in pixel values of the same column between consecutive rows in each channel. Averaging over rows and columns, we use only one representative number as d_k for each channel.

Considering an input vector $\vec{I} = (1, C, \vec{A}, \vec{D})$, the error metric of a full-sized output F is estimated by a linear regression model as:

$$F = \vec{I} \cdot w, \tag{9}$$

where w is a $(n+5) \times 1$ coefficient vector. In the experiment results, we will see that Model-CAD outperforms the other models.

3.5.4 Non-linear models.

We have also tested complex non-linear models to predict F, using artificial neural networks with all pixel information as input. However, considering the run-time complexity [23], we could not observe any significant benefit of the non-linear models over the linear models mentioned earlier. Thus, we do not report their results in the evaluation.

4 Implementation and Dataset

We use loop perforation and memoization [42, 30] to approximately filter the frames in the video. The implementation of VIDEOCHEF is comprised of an offline and an online component. The offline component uses a set of training videos (50% of videos described under the dataset below) and creates models for the error mapping and for the cost and the benefit of each step of the search. This last model is actually implemented as a lookup table, due to the space being only piece-wise continuous. During runtime, VIDEOCHEF queries these models, using linear interpolation if needed, and performs an efficient search to identify the optimal ALs and runs each of the three filters in any pipeline with their optimal values.

VIDEOCHEF API. Our compiler pass identifies the approximable blocks using program annotations and then performs the relevant transformations to insert the approximation knobs to be tuned (such as `approx_level` in Sec. 3.1). We have provided support for similar annotations in other domain specific languages that we have built in the past and that helps to reduce the programmer burden [28]. The user can then use the following API calls to enable VIDEOCHEF in the video pipeline:

- *setCalibrationFrequency(f=”I-frame”)* : This will set how frequently VIDEOCHEF will search for the best approximation settings. The default value is VIDEOCHEF will trigger a search for every I-frame. If *f*=”*x*”, then VIDEOCHEF will search every *x*-th frame.
- *setQualityThreshold(b=”30”)* : This will set the (lower) PSNR threshold that the approximated pipeline must deliver. Default is 30 dB. VIDEOCHEF exposes to the user approximate versions of many filters from the FFmpeg library, with names like *deflate_approx*. The developer of VIDEOCHEF can register a callback with the video decoder using the call *void notifyIFrame(void *)*.

Video Dataset. We used 106 YouTube MPEG-4 videos for our evaluation. We used `libvideo`, a lightweight .NET library [24], to download the videos. The videos were collected from 8 different categories to cover a spectrum of different motion and color artifacts in the frames: Lectures, Ads, Car Races, Entertainment, Movie trailers, Nature, News, and Sports. At the first step, a single seed video was downloaded from each category, then we downloaded all YouTube's recommendations to the seed video, which turned out to belong to the same category as that of the seed video. Once the set of videos was collected, we randomly sub-sampled a 20 second clip from each video, being motivated by a desire to bound the experiment time. For each category, we collected approximately 25 videos and filtered out those with low resolution (since the quality threshold was likely already breached with the original video).

5 Evaluation

We describe our benchmarks first and then the four experiments to evaluate the macro properties of VIDEOCHEF and then its various components.

Benchmarks. We construct our benchmark by including different video processing pipelines. Each video processing pipeline consists of 3 consecutive filters, which are selected from a pool of 10 video filters from the FFmpeg library. These filters are modified to support approximation with tuning knobs. To execute on these filter pipelines, one needs to provide a video input and a quality threshold. Finally, the output is also a video, together with a quality metric with respect to each frame. We have a total of 9 different filter pipelines.

Quality Metric. We use PSNR (Eq. 1) as the quality metric for the videos produced by the approximate pipelines. We present the results for two acceptable PSNR thresholds. The threshold of 30 dB is considered a typical lower bound for lossy image and video compres-

Table 1: Summary of the analyzed pipelines. We denote the approximation applied to each filter: Loop Perforation (LP) or Memoization (M)

Name	Description, labeled with Approx. type	Approximation Type	Approximation Levels
DEB	Deflate(LP)-Emboss(LP)-Boxblur(M)	Loop perforation(LP) & Memoization(M)	1-6, 1-6, 1-6
DVE	Deflate(LP)-Vignette(LP)-Emboss(LP)	Loop perforation	1-6, 1-6, 1-6
BVI	Boxblur(M)-Vignette(LP)-Inflate(LP)	Loop perforation & Memoization	1-6, 1-6, 1-6
UIV	Unsharp(LP)-Inflate(LP)-Vignette(LP)	Loop perforation	1-6, 1-6, 1-6
DUE	Dilation(LP)-Unsharp(LP)-Emboss(LP)	Loop perforation	1-6, 1-6, 1-6
BVD	Boxblur(M)-Vignette(LP)-Dilation(LP)	Loop perforation & Memoization	1-6, 1-6, 1-6
UEE	Unsharp(LP)-Erosion(LP)-Emboss(LP)	Loop perforation	1-6, 1-6, 1-6
EUB	Erosion(LP)-Unsharp(LP)-Boxblur(M)	Loop perforation & Memoization	1-6, 1-6, 1-6
BUC	Boxblur(M)-Unsharp(LP)-Colorbalance(LP)	Loop perforation & Memoization	1-6, 1-6, 1-6

sion [48, 16]. The threshold of 20 dB is considered the lower bound for lossy wireless transmission [44].

Evaluation Metrics. We define improvement as decrease in execution time, expressed as a percentage of the competitive protocol. We define the speedup of our approach as $Speedup = \frac{\textit{Speed of our protocol}}{\textit{Speed of compared protocol}} - 1$

Setup. We split the input videos into three groups: training, validation and test, with a share of 50%, 25%, and 25% of the videoset. The experiments are done on an x86 server with a six-core Intel(R) Xeon CPU, 16 GB RAM, and Ubuntu Linux kernel 4.4. We used FFmpeg library version 3.0 (compiled with gcc 5.4.0).

Canary Input Selection. We fixed the canary size to be $1/64$ of the original size because our preliminary experiments showed that it is a good parameter to guarantee a dissimilarity metric value of 10% or less. Thus, the overhead of appropriate canary selection is not included in the results.

5.1 Performance and Quality Comparison for End-to-End Workflow

Figure 3 presents the results of the end-to-end workflow for the nine different video processing pipelines over all videos from the test set. Each plot presents the speedup relative to the exact pipeline for the following configurations (from left to right):

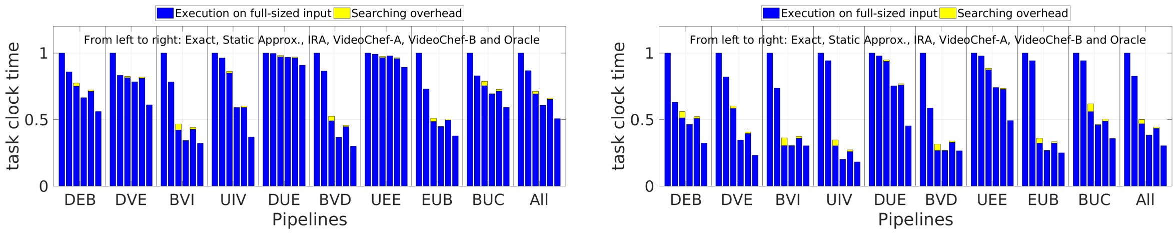
- Exact computation, with default parameters.
- Best static approximation, created by setting the AL that is just over the error threshold for *all* the frames in training videos.
- IRA extended with a simple searching policy that has a fixed interval of 10 frames. This number is chosen according to SAGE [37], which gives an analytic bound for a video processing setting.
- VIDEOCHEF version A – with I-frames detection.
- VIDEOCHEF version B – with scene change detection.
- Oracle version uses exhaustive search but does not incur search overhead. This sets the upper bound of the performance.

For both VIDEOCHEF versions, we used the CAD error model with 3dB margin, as the result of the analysis in Section 5.3.

Performance for 30db Threshold. Figure 3(a) shows that VIDEOCHEF version A reduces the execution time by 39.1% over exact computation and is within 20% of the Oracle. It outperforms both static approximation and IRA, by respectively 29.9% and 14.6% in the aggregate. The advantage exists for all the video filter pipelines with the greatest savings relative to IRA being in Unsharp-Inflate-Vignette (UIV) pipeline. We are 39.2%, 36.8% and 29.5% better than exact computation, static approximation, and IRA, respectively. The search overhead for VIDEOCHEF (both versions A and B) is small – the yellow portions of the bars are almost not visible – and yet it finds more aggressive approximations than the competitive approaches (static or IRA) (the blue portions of the bars are shorter). The IRA approach, due to its assumption that the error in the canary output is identical to the error in the full output, cannot use aggressive ALs and thus cannot achieve the full speedup available through approximation. Within the two variants of VIDEOCHEF, scene change detector (version B) is slower than an I-frame lookup (version A).

Performance for 20db Threshold. We also evaluate on VIDEOCHEF on a different quality thresholds 20dB. Given a larger error budget, Figure 3 shows that VIDEOCHEF is able to achieve more performance gain over exact computation (1.6x speedup). We also outperform static approximation and IRA by 53.4% and 23.1% and within 26.6% from the Oracle results. Notice that the pipelines where we achieve the maximum performance gain over IRA changes from UIV to DVE.

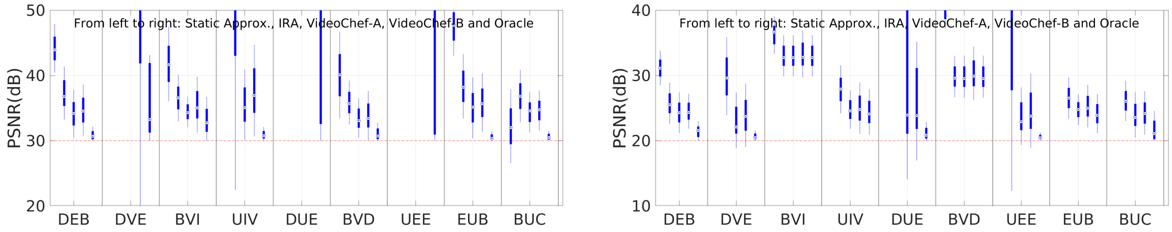
Quality for 30db Threshold. Figure 4(a) shows that IRA and static approximation both achieve much higher quality than what the user specified (30 dB), an undesirable outcome here since this comes at the expense of higher execution time. VIDEOCHEF on the other hand tracks the Oracle quality quite closely, which in turn meets the user requirement. It does however, drop below the threshold on some inputs, albeit by small amounts. This indicates that a future design feature should compensate for the tendency of VIDEOCHEF to sometime drop below the target video quality, say by adding a penalty function when the AL brings it close to the boundary. Further, a carefully designed margin in the searching algorithm can reduce the violation in quality



(a) Quality threshold = 30dB

(b) Quality threshold = 20dB

Figure 3: Mean execution times over all frames of all videos. Geometric means of the speedups are on the right.



(a) Quality threshold = 30dB

(b) Quality threshold = 20dB

Figure 4: Quality of each frame across different video filter pipelines.

requirement but still achieve speedup. The careful reader would have noticed that for some pipelines, some protocol results are missing here. This happens because no approximation is possible for some pipelines and there is no error introduced and hence, PSNR is not defined.

We also use the percentage of frames that violate the quality threshold to characterize the robustness of each protocol. The violation rate of static approximation, IRA, VIDEOCHEF version A and B are 3.27%, 0.64%, 6.6% and 4.79%. Although the two versions of VIDEOCHEF have higher violation rates, they are still within a typical user acceptable threshold (5%). We consider the violation may due to two factors – (1) Inaccurate error prediction in the key frame. (2) The quality of non-key frames degrade and drop below the threshold before a fresh key frame is identified and a search triggered. According to our modeling in Sec 5.3, the violation due to the first factor is limited to at most 5%, while the second error may be inevitable as long as we do not search for every frame. Considering the tradeoff between searching overhead and better error control, VIDEOCHEF is able to largely reduce the searching overhead and still maintain good quality.

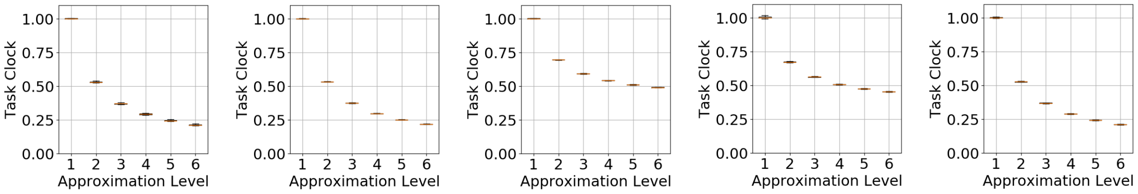
Quality for 20db Threshold. Figure 4(b) shows the quality measurement of different protocols across all the pipelines. The mean violation rate averaged across all pipelines of static approximation, IRA, VIDEOCHEF version A and B are 0%, 0.23%, 7.18% and 3.93%. In the two quality threshold case, we see the advantage of scene change detection as an add-on in VIDEOCHEF version B to decrease the violation rate because it can accurately detect the frame which differs largely from the previous and trigger a required search for optimal approximation levels.

5.2 Speedup and Video Quality versus Approximation Levels

This experiment studies (1) how the execution time of each filter varies with the AL setting for that filter and (2) how the video quality varies with the AL setting. This result is dependent on the approximation technique but is independent of the VIDEOCHEF configuration used to decide on the AL. We show the results with all the videos in our dataset and 5 out of 10 representative filters in Figure 5 (number of executed instructions) and Figure 6 (video quality). When showing the result for a specific filter, we only execute on this filter and not the 3-stage pipeline. Here the results have higher variability due to the content-dependent effect. For the execution time, we normalize by the measure for exact computation.

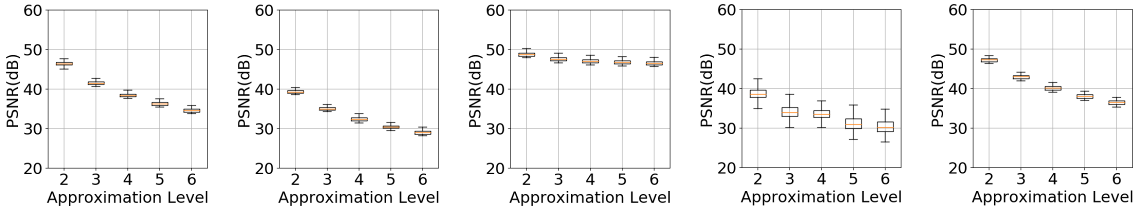
Execution Time. Figure 5 shows that as the AL becomes higher, *i.e.*, the approximation becomes more aggressive, the execution time decreases. But the rate of decrease slows down as the AL becomes higher and the behaviors among the different filters in our evaluation are comparable. Note that this is a box plot, but there is little variation across the different videos and hence each AL gives very tight result. This is expected because the amount of processing done in the filter, whether with exact or approximate computation, is *not* content dependent, but the effect of the approximation *is* content dependent.

Quality. Figure 6 shows the effect of AL on the video quality when the full frame is used. The quality degrades as the approximation gets more aggressive, but the nature of the decrease is not uniform across all the filters. Even within each filter, the effect on quality depends on the ex-



(a) Deflate filter (b) Emboss filter (c) Boxblur filter (d) Histeq filter (e) Vignette filter

Figure 5: The normalized execution time for each filter as the Approximation Level (AL) is varied, across all 106 videos in our dataset. The number of CPU cycles is normalized by the measure for exact computation. As the AL increases, the execution time decreases and this happens consistently across all videos and filters.



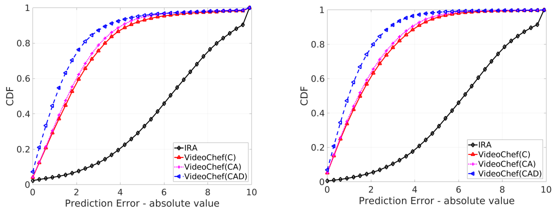
(a) Deflate filter (b) Emboss filter (c) Boxblur filter (d) Histeq filter (e) Vignette filter

Figure 6: The video quality for each filter as the Approximation Level (AL) is varied, across all 106 videos in our dataset. The effect depends on the video content and the filter being used.

act video frame, as implied by the vertical data spread for any given AL. We identify two forms of unpredictability of how AL correlates with video quality: with the content (which video frame is being approximated) and with the filter. Due to these two factors, we do not try to come up with a closed form curve for doing the prediction, rather, we do the actual computation with the canary input for a given AL setting, compute the PSNR, and then map it to the PSNR with the full input (Section 3.5). Contrast this to the execution time where we create a lookup table through training, which is content independent, and just look it up during the online search (Section 3.4.1). The variability due to video content in the PSNR plot validates our rationale for doing the approximation in a content-dependent manner. The rationale is shared with [25], but it sets our work apart from the approximation techniques that select the approximation configuration in a content-independent manner.

5.3 Evaluation of Error Mapping Models

In this experiment, we evaluate the quality of the various error mapping models in VIDEOCHEF. We trained the model on the training video set. Table 2 and Figure 3 present the performance of our model on the validation videos. Figure 7 shows that even a simple model C can greatly reduce the prediction error relative to IRA. Also, as we increase the level of knowledge, the model achieves higher prediction accuracy and model-CAD performs the best due to its good use of the feature extraction from the frames. We can see that with our CAD model, we can successfully control the error within 2dB in 80% of the cases and within 3dB in 90% of the cases. Given these results, we set up a 3dB margin when mapping from the canary error to the full error.



(a) Validation set (b) Test set

Figure 7: Results of the error modeling in VIDEOCHEF mapping error in canary output to error in full output. The CAD model with characteristics of the frame performs best, though it is only slightly better than the CA models.

Table 2: F-1 measure of different error mapping models averaged over all pipelines. We regard IRA as a pass through error mapping.

Models	IRA	C	CA	CAD
30dB threshold	0.8650	0.9576	0.9594	0.9686
20dB threshold	0.8007	0.9679	0.9660	0.9759

The results on the test videos (Section 5.1) show that the cases when we violate the quality requirement are within 10%.

5.4 User Perception Study

To evaluate if the protocols cause any perceptual difference, we conduct a small user study with 16 participants. Users were recruited by emailing students of certain ECE classes. We picked 16 videos, 2 from each YouTube content category, randomly picked from our dataset. We processed each video (a snippet of 20 seconds from each, as in the rest of the evaluation) using the Oracle approach and using VIDEOCHEF for pipeline DBE.

Table 3: Results of the user studies with 16 videos processed using Oracle and VIDEOCHEF

Degree of difference	Percentage
No difference	58.59%
Little difference	34.77%
Large difference	6.64%
Total difference	0

This pipeline was chosen because its result in the rest of the evaluation is representative and it produces videos which are still visually pleasing. In the experiment, we showed the two versions of each of the 16 videos concurrently, processed using the Oracle and VIDEOCHEF tools, without letting the participant know which window corresponded to which tool. All participants watched the videos independently. The participants were asked to rate the videos in four categories: Same, Little difference, Large difference, and Total difference. We gave guidance to the participants for the four categories as difference ∈ [0%,5%),[5%,20%),[20%,50%), and ≥ 50%.

We show the results in Table 3. The percentage figure is the percentage of the total number of videos shown, which is 16 × 16 (number of videos × number of users). We conclude that 58.59% of the videos got no difference rating between the Oracle and the VIDEOCHEF processed videos, while 34.77% got a little difference rating. Although 6.64% of videos got large difference rating, none of the videos got total difference rating. This validates that qualitatively human perception is not seeing significant difference in video quality due to approximate processing using VIDEOCHEF.

6 Related Work

Approximate Tradeoffs in Computations and Data. Researchers presented various techniques for changing computations at the system level to trade accuracy for performance, *e.g.*, in hardware [32, 47, 12, 11, 8], runtime systems [3, 18], and compilers [30, 42, 2, 39, 6]. A key challenge of approximate computing is finding good tradeoffs between accuracy and performance. For this, researchers have looked at both off-line autotuning [30, 42, 29, 38] and on-line dynamic adaptation [3, 18, 37, 22, 17]. In image processing, various techniques exist for synthesizing approximate filter versions, e.g., using genetic programming [45, 41, 13]. Recently, Lou et al. [26] present "image perforation", an adaptive version of loop perforation tailored for individual image filters. Researchers also proposed storing multimedia data in approximate memories, including standard [39, 34], solid-state [40], and multi-level cell memories specialized for video encodings [19]. We consider such storage approaches complementary to our computation-based technique for video encoding.

Input-Aware Approximation. Several techniques provide input-aware approximations to monitor output quality and control the aggressiveness of the approximation during execution. Green [3] was an early approach that applied dynamic quality monitoring to adjust the level of approximation, based on a user-defined quality function. More recently, input-aware approximation identifies classes of similar inputs and applies different approximations for each input class [9, 43]. Opprox [31] learns the control-flow of the input-optimized program and then selects in which phase to approximate as well as how much to approximate. In contrast to our work, all these approaches use off-line models for prediction of input quality and do not craft the smaller inputs at runtime. Ringenburg *et. al.* [35] proposed online monitoring mechanisms, where a random subset of approximate outputs is compared with a precise output on a sampling basis, or the output of the current execution is predicted from past executions with similar inputs. Raha *et al.* [33] present a precise analysis of accuracy for a commonly used reduce-and-rank computational pattern. Rumba [22] and Topaz [1] detect outliers in intermediate computation results. In contrast to IRA [25] and our VIDEOCHEF, these approaches do not use canary inputs to guide the optimization and monitoring and therefore grapple with the overhead issue.

7 Conclusion

Fast and resource efficient processing of videos is required in many scenarios. We built a resource efficient and input-aware approximate video processing pipeline called VIDEOCHEF. VIDEOCHEF controls the approximation in each frame (using the properties of the frame) to meet the user's accuracy requirement. In particular, VIDEOCHEF uses a canary-input based approach for fast searching, as proposed in prior work, but overcomes some fundamental challenges by innovating a machine-learning based accurate error estimation technique and an input-aware search technique that finds best approximation settings. We show that VIDEOCHEF can provide significant speedup in 9 different video processing pipelines while satisfying user's quality requirements.

8 Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS-1527262 and CCF-1703637 and by Sandia National Lab under their Academic Alliance Program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors. We thank Ashraf Mahgoub, Manish Gupta, Edward J. Delp, Fengqing Maggie Zhu, and Di Chen for useful discussions and advice.

References

[1] Achour, S., and Rinard, M. C. Approximate computation with outlier detection in topaz. In *OOPSLA* (2015).

[2] Ansel, J., Wong, Y. L., Chan, C., Olszewski, M., Edelman, A., and Amarasinghe, S. Language and compiler support for auto-tuning variable-accuracy algorithms. In *CGO* (2011).

[3] Baek, W., and Chilimbi, T. M. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI* (2010).

[4] Bao, L., Yang, Q., and Jin, H. Fast edge-preserving patchmatch for large displacement optical flow. In *IEEE CVPR* (2014), pp. 3534–3541.

[5] Buckler, M., Jayasuriya, S., and Sampson, A. Reconfiguring the imaging pipeline for computer vision. In *The IEEE International Conference on Computer Vision (ICCV)* (2017).

[6] Carbin, M., Misailovic, S., and Rinard, M. C. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA* (2013).

[7] Chaudhuri, S., Gulwani, S., Lublinerman, R., and Navidpour, S. Proving programs robust. In *FSE* (2011).

[8] Chippa, V. K., Mohapatra, D., Raghunathan, A., Roy, K., and Chakradhar, S. T. Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency. In *DAC* (2010).

[9] Ding, Y., Ansel, J., Veeramachaneni, K., Shen, X., O'Reilly, U.-M., and Amarasinghe, S. Autotuning algorithmic choice for input sensitivity. In *PLDI* (2015).

[10] Dougherty, E. R., and Lotufo, R. A. *Hands-on morphological image processing*, vol. 59. SPIE press, 2003.

[11] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Architecture support for disciplined approximate programming. In *ASPLOS* (2012).

[12] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Neural acceleration for general-purpose approximate programs. In *MICRO* (2012), IEEE Computer Society.

[13] Farbman, Z., Fattal, R., and Lischinski, D. Convolution pyramids. *ACM Trans. Graph. 30*, 6 (2011), 175–1.

[14] Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G., and Winstein, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI* (2017), pp. 363–376.

[15] Goiri, Í., Bianchini, R., Nagarakatte, S., and Nguyen, T. D. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ASPLOS* (2015).

[16] Hamzaoui, R., and Saupe, D. *Fractal image compression - in "Document and Image Compression"*. CRC Press, 2006.

[17] Hoffmann, H. Jouleguard: energy guarantees for approximate applications. In *SOSP* (2015).

[18] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. Dynamic knobs for responsive power-aware computing. In *ASPLOS* (2011).

[19] Jevdjic, D., Strauss, K., Ceze, L., and Malvar, H. S. Approximate storage of compressed and encrypted videos. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ACM, pp. 361–373.

[20] Jiang, H., Helal, A. S., Elmagarmid, A. K., and Joshi, A. Scene change detection techniques for video database systems. *Multimedia Systems 6*, 3 (May 1998), 186–195.

[21] Juurlink, B., Alvarez-Mesa, M., Chi, C. C., Azevedo, A., Meenderinck, C., and Ramirez, A. Understanding the application: An overview of the h. 264 standard. *Scalable Parallel Programming Applied to H. 264/AVC Decoding* (2012), 5–15.

[22] Khudia, D. S., Zamirai, B., Samadi, M., and Mahlke, S. Rumba: an online quality management system for approximate computing. In *ISCA* (2015).

[23] Kim, S. G., Harwani, M., Grama, A., and Chaterji, S. Ep-dnn: A deep neural network-based global enhancer prediction algorithm. *Scientific reports 6* (2016), 38433.

[24] Ko, J. Libvideo: A lightweight .net library to download youtube videos. https://github.com/jamesqo/libvideo, 2016.

[25] Laurenzano, M. A., Hill, P., Samadi, M., Mahlke, S., Mars, J., and Tang, L. Input responsiveness: using canary inputs to dynamically steer approximation. In *PLDI* (2016), ACM.

[26] Lou, L., Nguyen, P., Lawrence, J., and Barnes, C. Image perforation: Automatically accelerating image pipelines by intelligently skipping samples. *ACM Transactions on Graphics (TOG) 35*, 5 (2016), 153.

[27] Mahadik, K., Chaterji, S., Zhou, B., Kulkarni, M., and Bagchi, S. Orion: Scaling genomic sequence matching with fine-grained parallelization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), IEEE Press, pp. 449–460.

[28] Mahadik, K., Wright, C., Zhang, J., Kulkarni, M., Bagchi, S., and Chaterji, S. Sarvavid: a domain specific language for developing scalable computational genomics applications. In *Proceedings of the 2016 International Conference on Supercomputing* (2016), ACM, p. 34.

[29] Meng, J., Chakradhar, S., and Raghunathan, A. Best-effort parallel execution framework for recognition and mining applications. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (2009), IEEE, pp. 1–12.

[30] Misailovic, S., Sidiroglou, S., Hoffmann, H., and Rinard, M. Quality of service profiling. In *ICSE* (2010).

[31] Mitra, S., Gupta, M. K., Misailovic, S., and Bagchi, S. Phase-aware optimization in approximate computing. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO)* (2017), IEEE Press, pp. 185–196.

[32] Palem, K. V. Energy aware computing through probabilistic switching: A study of limits. *IEEE Transactions on Computers 54*, 9 (2005), 1123–1137.

[33] Raha, A., Venkataramani, S., Raghunathan, V., and Raghunathan, A. Quality configurable reduce-and-rank for energy efficient approximate computing. In *DATE* (2015).

[34] Ranjan, A., Raha, A., Venkataramani, S., Roy, K., and Raghunathan, A. Aslan: Synthesis of approximate sequential circuits. In *DATE* (2014).

[35] Ringenburg, M., Sampson, A., Ackerman, I., Ceze, L., and Grossman, D. Monitoring and debugging the quality of results in approximate programs. In *ASPLOS* (2015).

[36] Samadi, M., Jamshidi, D. A., Lee, J., and Mahlke, S. Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS* (2014).

[37] Samadi, M., Lee, J., Jamshidi, D. A., Hormati, A., and Mahlke, S. Sage: Self-tuning approximation for graphics engines. In *MICRO* (2013).

[38] Sampson, A., Baixo, A., Ransford, B., Moreau, T., Yip, J., Ceze, L., and Oskin, M. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01* (2015).

[39] Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. Enerj: Approximate data types for safe and general low-power computation. In *PLDI* (2011).

[40] Sampson, A., Nelson, J., Strauss, K., and Ceze, L. Approximate storage in solid-state memories. *ACM TOCS* (2014).

[41] Sharman, K., Alcazar, A., and Li, Y. Evolving signal processing algorithms by genetic programming. In *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)* (1995), IET, pp. 473–480.

[42] Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., and Rinard, M. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE* (2011).

[43] Sui, X., Lenharth, A., Fussell, D. S., and Pingali, K. Proactive control of approximate programs. In *ASPLOS* (2016).

[44] Thomos, N., Boulgouris, N. V., and Strintzis, M. G. Optimized transmission of jpeg2000 streams over wireless channels. *IEEE Transactions on image processing 15*, 1 (2006), 54–67.

[45] Uesaka, K., and Kawamata, M. Evolutionary synthesis of digital filter structures using genetic programming. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing 50*, 12 (2003), 977–983.

[46] Union, E. General data protection regulation. http://www.eugdpr.org/, 2017.

[47] Venkataramani, S., Chippa, V. K., Chakradhar, S. T., Roy, K., and Raghunathan, A. Quality programmable vector processors for approximate computing. In *MICRO* (2013).

[48] Welstead, S. T. *Fractal and Wavelet Image Compression Techniques*, 1st ed. Society of Photo-Optical Instrumentation Engineers (SPIE), Bellingham, WA, USA, 1999.

[49] Zhang, H., Ananthanarayanan, G., Bodik, P., Philipose, M., Bahl, P., and Freedman, M. J. Live video analytics at scale with approximation and delay-tolerance. In *NSDI* (2017), pp. 377–392.

[50] Zhang, T., Chowdhery, A., Bahl, P. V., Jamieson, K., and Banerjee, S. The design and implementation of a wireless video surveillance system. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (2015), ACM, pp. 426–438.

SOCK: Rapid Task Provisioning with Serverless-Optimized Containers

Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter[†],
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin–Madison [†] Microsoft Gray Systems Lab

Abstract

Serverless computing promises to provide applications with cost savings and extreme elasticity. Unfortunately, slow application and container initialization can hurt common-case latency on serverless platforms. In this work, we analyze Linux container primitives, identifying scalability bottlenecks related to storage and network isolation. We also analyze Python applications from GitHub and show that importing many popular libraries adds about 100 ms to startup. Based on these findings, we implement SOCK, a container system optimized for serverless workloads. Careful avoidance of kernel scalability bottlenecks gives SOCK an 18× speedup over Docker. A generalized-Zygote provisioning strategy yields an additional 3× speedup. A more sophisticated three-tier caching strategy based on Zygotes provides a 45× speedup over SOCK without Zygotes. Relative to AWS Lambda and OpenWhisk, OpenLambda with SOCK reduces platform overheads by 2.8× and 5.3× respectively in an image processing case study.

1. Introduction

The effort to maximize developer velocity has driven many changes in the way programmers write and run their code [43]. Programmers are writing code at a higher level of abstraction: JavaScript, Python, Java, Ruby, and PHP are now the most popular languages on GitHub (in that order), surpassing lower-level languages such as C and C++ [51]. Developers also increasingly focus on application-specific logic, reusing existing libraries for general functionality when possible [19, 23, 40].

New programming paradigms are also liberating developers from the distraction of managing servers [18, 52, 54]. In particular, a proliferation of new serverless platforms [5, 6, 14, 17, 20, 39, 45] allow developers to construct applications as a set of handlers, called *lambdas*, commonly written in Python (or some other high-level language), that execute in response to events, such as web requests or data generation. Serverless providers automatically scale the number of handlers up and down to accommodate load so that developers need not worry about the number or configuration of machines serving their workload. Using serverless platforms is often very economical: billing granularity is in fractions of a second, and there is generally no tenant charge for idle time.

These three strategies (*i.e.*, programming at higher abstraction levels, reusing libraries, and decomposing applications into auto-scaling serverless lambdas) improve developer velocity, but they also create new infrastructure problems. Specifically, these techniques make process cold-start more expensive and frequent. Languages such as Python and JavaScript require heavy runtimes, making startup over 10× slower than launching an equivalent C program [1]. Reusing code introduces further startup latency from library loading and initialization [4, 8, 26, 27]. Serverless computing amplifies these costs: if a monolithic application is decomposed to N serverless lambdas, startup frequency is similarly amplified. Lambdas are typically isolated from each other via containers, which entail further sandboxing overheads [31].

Fast cold start is important for both tenants and providers. A graceful reaction to flash crowds [15, 22] requires concurrent low-latency deployment to many workers. From a provider perspective, avoiding cold starts can be quite costly. Most serverless platforms currently wait minutes or hours to recycle idle, unbilled lambda instances [50]. If cold start is made faster, providers will be able to reclaim idle resources and rebalance load across machines more aggressively.

In order to better understand the sandboxing and application characteristics that interfere with efficient cold start, we perform two detailed studies. First, we analyze the performance and scalability of various Linux isolation primitives. Among other findings, we uncover scalability bottlenecks in the network and mount namespaces and identify lighter-weight alternatives. Second, we study 876K Python projects from GitHub and analyzing 101K unique packages from the PyPI repository. We find that many popular packages take 100 ms to import, and installing them can take seconds. Although the entire 1.5 TB package set is too large to keep in memory, we find that 36% of imports are to just 0.02% of packages.

Based on these findings, we implement SOCK (roughly for serverless-optimized containers), a special-purpose container system with two goals: (1) *low-latency invocation* for Python handlers that import libraries and (2) *efficient sandbox initialization* so that individual workers can achieve high steady-state throughput. We integrate SOCK with the OpenLambda [20] serverless platform, replacing Docker as the primary sandboxing mechanism.

SOCK is based on three novel techniques. First, SOCK uses lightweight isolation primitives, avoiding the performance bottlenecks identified in our Linux primitive study, to achieve an 18× speedup over Docker. Second, SOCK provisions Python handlers using a generalized Zygote-provisioning strategy to avoid the Python initialization costs identified in our package study. In the simplest scenarios, this technique provides an additional 3× speedup by avoiding repeated initialization of the Python runtime. Third, we leverage our generalized Zygote mechanism to build a three-tiered package-aware caching system, achieving 45× speedups relative to SOCK containers without Zygote initialization. In an image-resizing case study, SOCK reduces cold-start platform overheads by 2.8× and 5.3× relative to AWS Lambda and OpenWhisk, respectively.

The rest of this paper is structured as follows. We study the costs of Linux provisioning primitives (§2) and application initialization (§3), and use these findings to guide the design and implementation of SOCK (§4). We then evaluate the performance of SOCK (§5), discuss related work (§6), and conclude (§7).

2. Deconstructing Container Performance

Serverless platforms often isolate lambdas with containers [14, 20, 39, 45]. Thus, optimizing container initialization is a key part of the lambda cold-start problem. In Linux, containerization is not a single-cohesive abstraction. Rather, general-purpose tools such as Docker [36] are commonly used to construct containers using a variety of Linux mechanisms to allocate storage, isolate resources logically, and isolate performance. The flexibility Linux provides also creates an opportunity to design a variety of special-purpose container systems. In this section, we hope to inform the design of SOCK and other special-purpose container systems by analyzing the performance characteristics of the relevant Linux abstractions. In particular, we ask *how can one maximize density of container file systems per machine? What is the cost of isolating each major resource with namespaces, and which resources must be isolated in a serverless environment?* And *how can the cost of repeatedly initializing cgroups to isolate performance be avoided?* We perform our analysis on an 8-core m510 machine [11] with the 4.13.0-37 Linux kernel.

2.1 Container Storage

Containers typically execute using a root file system other than the host's file system. This protects the host's data and provides a place for the container's unique dependencies to be installed. Provisioning a file system for a container is a two step procedure: (1) populate a subdirectory of the host's file system with data and code needed by the container, and (2) make the subdirectory the root of the new container, such that code in the container can no longer access other host data. We explore alternative mechanisms for these population and access-dropping steps.

Populating a directory by physical copying is prohibitively slow, so all practical techniques rely on logical copying. Docker typically uses union file systems (*e.g.*, AUFS) for this purpose; this provides a flexible layered-composition mechanism and gives running containers copy-on-write access over underlying data. A simpler alternative is bind mounting. Bind mounting makes the same directory visible at multiple locations; there is no copy-on-write capability, so data that must be protected should only be bind-mounted as read-only in a container. To compare binds to layered file systems, we repeatedly mount and unmount from many tasks in parallel. Figure 1 shows the result: at scale, bind mounting is about twice as fast as AUFS.

Once a subdirectory on the host file system has been populated for use as a container root, the setup process must switch roots and drop access to other host file data. Linux provides two primitives for modifying the file-system visible to a container. The older `chroot` operation simply turns a subdirectory of the original root file system into the new root file system. A newer mount-namespace abstraction enables more interesting transformations: an `unshare` call (with certain arguments) gives a container a new set of mount points, originally identical to the host's set. The container's mount points can then be modified with mount, unmount, and other calls. It is even possible to reorganize the mount namespace of a container such that the container may see file system Y mounted on file system X (the container's root) while the host may see X mounted on Y (the host's root). There are cases where this powerful abstraction can be quite helpful, but overusing mount namespace flexibility "may quickly lead to insanity," as the Linux manpages warn [32].

We measure the scalability of mount namespaces, with results shown in Figure 2. We have a variable number of long-lived mount namespaces persisting throughout the experiment (x-axis). We churn namespaces, concurrently creating and deleting them (concurrency is shown by different lines). We observe that churn performance scales poorly with the number of prior existing namespaces: as the number of host mounts grows large, the rate at which namespaces can be cloned approaches zero. We also evaluate `chroot` (not shown), and find that using it entails negligible overhead (`chroot` latency is $< 1\mu$s).

2.2 Logical Isolation: Namespace Primitives

We have already described how mount namespaces can be used to virtualize storage: multiple containers can have

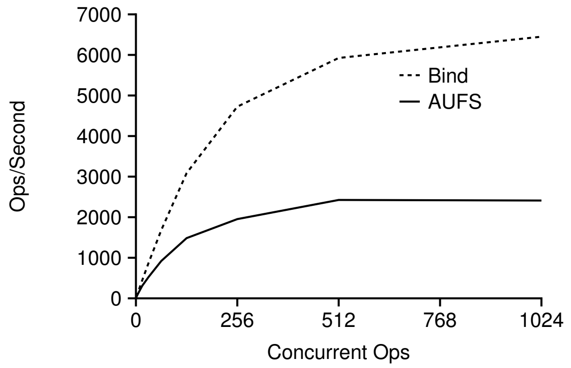


Figure 1. Storage Primitives. *The performance of mounting and unmounting AUFS file systems is compared to the performance of doing the same with bind mounts.*

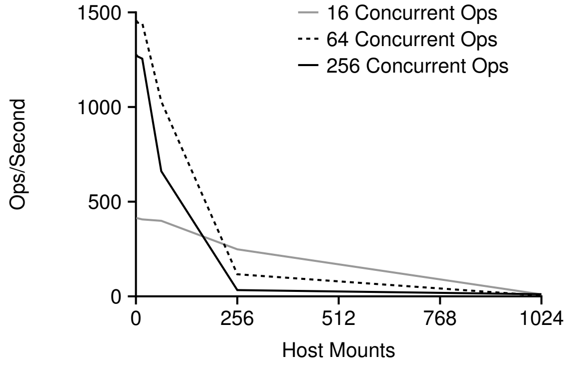


Figure 2. Mount Scalability. *This shows the rate at which processes can be unshared into new mount namespaces (y-axis) as the number of existing mounts varies (x-axis).*

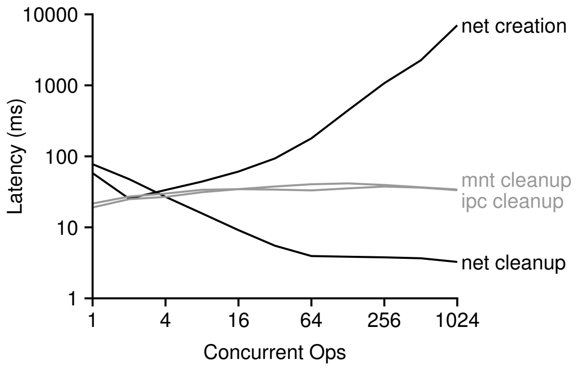


Figure 3. Namespace Primitives. *Lines show the latencies for copy_net_ns, put_mnt_ns and put_ipc_ns, and the average time spent by the network cleanup task per namespace.*

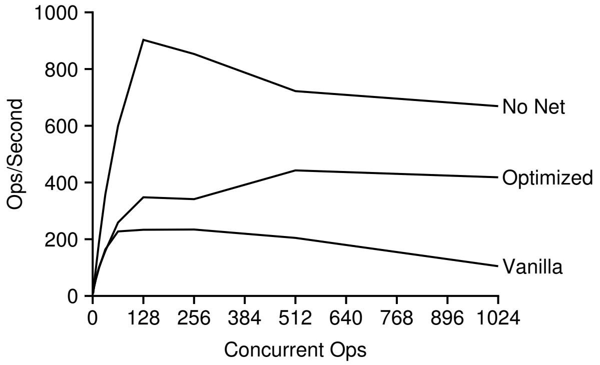


Figure 4. Network Namespace Performance. *A given number of containers (x-axis) are created and deleted in parallel with multiple network namespace configurations.*

access to their own virtual roots, backed by different physical directories in the host. Linux's network namespaces similarly allow different containers to use the same virtual port number (*e.g.*, 80), backed by different physical ports on the host (*e.g.*, 8080 and 8081). In this section, we study the collective use of mount and network namespaces, along with UTS, IPC, and PID namespaces [37] (user and cgroup namespaces are not evaluated here). The `unshare` call allows a process to create and switch to a new set of namespaces. Arguments to `unshare` allow careful selection of which resources need new namespaces. Namespaces are automatically reaped when the last process using them exits.

We exercise namespace creation and cleanup performance by concurrently invoking `unshare` and exiting from a variable number of tasks. We instrument the kernel with `ftrace` to track where time is going. Figure 3 shows the latency of the four most expensive namespace operations (other latencies not shown were relatively insignificant). We observe that mount and IPC namespace cleanup entails latencies in the tens of milliseconds. Upon inspection of the kernel code, we found that both operations are waiting for an RCU grace period [35]. During this time, no global locks are held and no compute is consumed, so these latencies are relatively harmless to overall throughput; as observed earlier (§2.1), it is possible to create ∼1500 mount namespaces per second, as long as churn keeps the number of namespaces small over time.

Network namespaces are more problematic for both creation and cleanup due to a single global lock that is shared across network namespaces [13]. During creation, Linux iterates over all existing namespaces while holding the lock, searching for namespaces that should be notified of the configuration change; thus, costs increase proportionally as more namespaces are created. As with mount and IPC namespaces, network-namespace cleanup requires waiting for an RCU grace period. However, for network namespaces, a global lock is held during that period, creating a bottleneck. Fortunately, network namespaces are cleaned in batches, so the per-namespace cost becomes small at scale (as indicated by the downward-sloping "net cleanup" line).

Figure 4 shows the impact of network namespaces on overall creation/deletion throughput (*i.e.*, with all

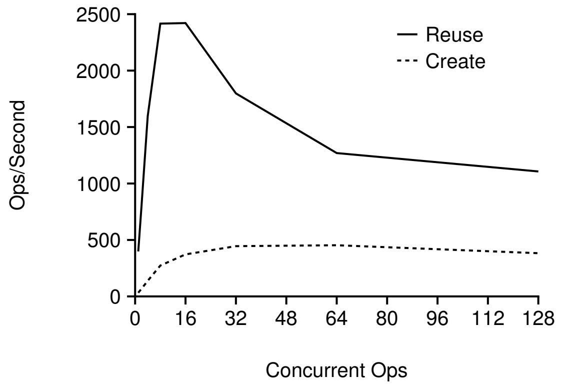


Figure 5. Cgroup Primitives. *Cgroup performance is shown for reuse and fresh-creation patterns.*

five namespaces). With unmodified network namespaces, throughput peaks at about 200 c/s (containers/second). With minor optimizations (disabling IPv6 and eliminating the costly broadcast code), it is possible to churn over 400 c/s. However, eliminating network namespaces entirely provides throughput of 900 c/s.

2.3 Performance Isolation: Cgroup Primitives

Linux provides performance isolation via the cgroup interface [9]. Processes may be assigned to cgroups, which are configured with limits on memory, CPU, file I/O, and other resources. Linux cgroups are easier to configure dynamically than namespaces. The API makes it simple to adjust the resource limits or reassign processes to different cgroups. In contrast, a mechanism for reassigning a process to a new PID namespace would need to overcome obstacles such as potential PID collisions.

The flexibility of cgroups makes two usage patterns viable. The first involves (1) creating a cgroup, (2) adding a process, (3) exiting the process, and (4) removing the cgroup; the second involves only Steps 2 and 3 (*i.e.*, the same cgroup is reused for different processes at different times). Figure 5 compares the cost of these two approaches while varying the numbers of tasks concurrently manipulating cgroups. Reusing is at least twice as fast as creating new cgroups each time. The best reuse performance is achieved with 16 threads (the number of CPU hyperthreads), suggesting cgroups do not suffer from the scaling issues we encountered with namespaces.

2.4 Serverless Implications

Our results have several implications for the design of serverless containers. First, in a serverless environment, all handlers run on one of a few base images, so the flexible stacking of union file systems may not be worth the the performance cost relative to bind mounts. Once a root location is created, file-system tree transformations that rely upon copying the mount namespace are costly at scale. When flexible file-system tree construction is not necessary, the cheaper `chroot` call may be used to drop access. Second, network namespaces are a major scalability bottleneck; while static port assignment may be useful in a server-based environment, serverless platforms such as AWS Lambda execute handlers behind a Network Address Translator [16], making network namespacing of little value. Third, reusing cgroups is twice as fast as creating new cgroups, suggesting that maintaining a pool of initialized cgroups may reduce startup latency and improve overall throughput.

3. Python Initialization Study

Even if lambdas are executed in lightweight sandboxes, language runtimes and package dependencies can make cold start slow [4, 8, 26, 27]. Many modern applications are accustomed to low-latency requests. For example, most Gmail remote-procedure calls are short, completing in under 100 ms (including Internet round trip) [20]. Of the short requests, the average latency is 27 ms, about the time it takes to start a Python interpreter and print a "hello world" message. Unless serverless platforms provide language- and library-specific cold-start optimizations, it will not be practical to decompose such applications into independently scaling lambdas. In this section, we analyze the performance cost of using popular Python libraries and evaluate the feasibility of optimizing initialization with caching. We ask: *what types of packages are most popular in Python applications? What are the initialization costs associated with using these packages?* And *how feasible is it to cache a large portion of mainstream package repositories on local lambda workers?*

3.1 Python Applications

We now consider the types of packages that future lambda applications might be likely to use, assuming efficient platform support. We scrape 876K Python projects from GitHub and extract likely dependencies on packages in the popular Python Package Index (PyPI) repository, resolving naming ambiguity in favor of more popular packages. We expect that few of these applications currently run as lambdas; however, our goal is to identify potential obstacles that may prevent them from being ported to lambdas in the future.

Figure 6 shows the popularity of 20 packages that are most common as GitHub project dependencies. Skew is high: 36% of imports are to just 20 packages (0.02% of the packages in PyPI). The 20 packages roughly fall into five categories: web frameworks, analysis, communication, storage, and development. Many of these use cases are likely applicable to future serverless applications. Current web frameworks will likely need to be replaced by serverless-oriented frameworks, but compute-intense analysis is ideal for lambdas [21]. Many lambdas will need libraries for communicating with other services and for storing data externally [16]. Development

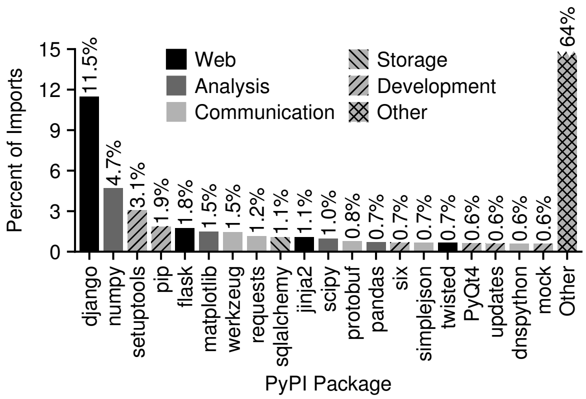


Figure 6. Package Popularity. *The twenty most used PyPI packages are shown. The bar labels represent the percentage of all GitHub-to-PyPI dependencies.*

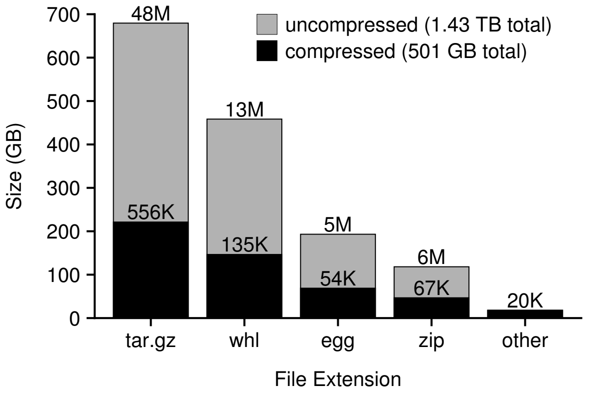


Figure 8. PyPI Package Data. *The size of the PyPI repository is shown, compressed and uncompressed, by file type (as of Mar 31, 2017). Bar labels show file counts.*

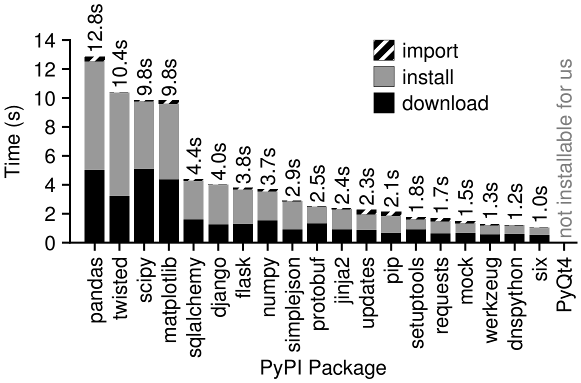


Figure 7. Startup Costs. *The download, install, and import times are shown for 20 popular Python packages, ordered by total initialization time.*

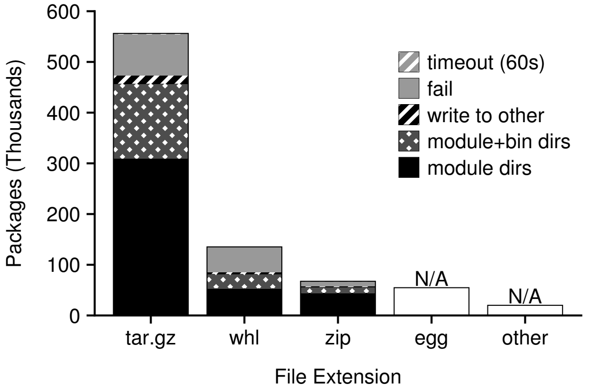


Figure 9. File-System Modifications. *The bars breakdown installations by the types of writes to the file system. The egg and other files can be used without extraction.*

libraries may be somewhat less relevant, but lambda-based parallel unit testing is an interesting use case.

If a package is being used for the first time, it will be necessary to *download* the package over the network (possibly from a nearby mirror), *install* it to local storage, and *import* the library to Python bytecode. Some of these steps may be skipped upon subsequent execution, depending on the platform. Figure 7 shows these costs for each of the popular packages. Fully initializing a package takes 1 to 13 seconds. Every part of the initialization is expensive on average: downloading takes 1.6 seconds, installing takes 2.3 seconds, and importing takes 107 ms.

3.2 PyPI Repository

We now explore the feasibility of supporting full language repositories locally on serverless worker machines. We mirror and analyze the entire PyPI repository, which contains 101K unique packages. Figure 8 shows the footprint of the entire repository, including every version of every package, but excluding indexing files. The packages are about 1.5 TB total, or ~0.5 TB compressed. Most packages are compressed as *.tar.gz* files or a zip-based format (*.whl*, *.egg*, or *.zip*). Across all format types, the average package contains about 100 files (*e.g.*, 135K *.whl* packages hold 13M compressed files).

We wish to understand how many of the PyPI packages could coexist when installed together. PyPI packages that unpack to a single directory can easily coexist with other installed packages, whereas packages that modify shared files may break other packages. We attempt to install every version of every PyPI package in its own Docker Ubuntu container (using a 1-minute timeout) and identify file creations and modifications. We ignore changes to temporary locations. Figure 9 shows the results for *.tar.gz*, *.whl*, and *.zip* distributions (*.egg* libraries are used directly without a prior installation, so we skip those). While fewer than 1% timed out, 18% simply failed to install in our container. 66% of succeeding installs only populate the local Python module directory (the *module dirs* category). Another 31% of succeeding installs modified just the module directories and the local bin directory (Python modules are sometimes bundled with various utilities). We conclude it is possible for 97% of installable packages to coexist in a single local install.

3.3 Serverless Implications

Downloading and installing a package and its dependencies from a local mirror takes seconds; furthermore, import of installed packages takes over 100 ms. Fortunately, our analysis indicates that storing large package repositories locally on disk is feasible. Strong popularity skew further creates opportunities to pre-import a subset of packages into interpreter memory [8].

4. SOCK with OpenLambda

In this section, we describe the design and implementation of SOCK, a container system optimized for use in serverless platforms. We integrate SOCK with the OpenLambda serverless platform, replacing Docker containers as the primary sandboxing mechanism for OpenLambda workers and using additional SOCK containers to implement Python package caching. We design SOCK to handle high-churn workloads at the worker level. The local churn may arise due to global workload changes, rebalancing, or aggressive reclamation of idle resources.

SOCK is based on two primary design goals. First, we want *low-latency invocation* for Python handlers that import libraries. Second, we want *efficient sandbox initialization* so that individual workers can achieve high steady-state throughput. A system that hides latency by maintaining pools of pre-initialized containers (*e.g.*, the LightVM approach [31]) would satisfy the first goal, but not the second. A system that could create many containers in parallel as part of a large batch might satisfy the second goal, but not the first. Satisfying both goals will make a serverless platform suitable for many applications and profitable for providers.

Our solution, SOCK, takes a three-pronged approach to satisfying these goals, based on our analysis of Linux containerization primitives (§2) and Python workloads (§3). First, we build a lean container system for sandboxing lambdas (§4.1). Second, we generalize Zygote provisioning to scale to large sets of untrusted packages (§4.2). Third, we design a three-layer caching system for reducing package install and import costs (§4.3).

4.1 Lean Containers

SOCK creates lean containers for lambdas by avoiding the expensive operations that are only necessary for general-purpose containers. Creating a container involves constructing a root file system, creating communication channels, and imposing isolation boundaries. Figure 10 illustrates SOCK's approach to these three tasks.

Storage: Provisioning container storage involves first populating a directory on the host to use as a container root. Bind mounting is faster using union file systems (§2.1), so SOCK uses bind mounts to stitch together a root from four host directories, indicated by the "F" label in Figure 10. Every container has the same Ubuntu base for its root file system ("base"); we can afford to back this by a RAM disk as every handler is required to use the same base. A packages directory used for package caching ("packages") is mounted over the base, as described later (§4.3). The same base and packages are read-only shared in every container. SOCK also binds handler code ("λ code") as read-only and a writable scratch directory ("scratch") in every container.

Once a directory has been populated as described, it should become the root directory. Tools such as Docker accomplish this by creating a new mount namespace, then restructuring it. We use the faster and simpler `chroot` operation (§2.1) since it is not necessary to selectively expose other host mounts within the container for serverless applications. SOCK containers always start with two processes ("init" and "helper" in Figure 10); both of these use `chroot` during container initialization, and any children launched from these processes inherit the same root.

Communication: The scratch-space mount of every SOCK container contains a Unix domain socket (the black pentagon in Figure 10) that is used for communication between the OpenLambda manager and processes inside the container. Event and request payloads received by OpenLambda are forwarded over this channel.

The channel is also used for a variety of control operations (§4.2). Some of these operations require privileged access to resources not normally accessible inside a container. Fortunately, the relevant resources (*i.e.*, namespaces and container roots) may be represented as file descriptors, which may be passed over Unix domain sockets. The manager can thus pass specific capabilities over the channel as necessary.

Isolation: Linux processes may be isolated with a combination of cgroup (for performance isolation) and namespace primitives (for logical isolation). It is relatively expensive to create cgroups; thus, OpenLambda creates a pool of cgroups (shown in Figure 10) that can be used upon SOCK container creation; cgroups are returned to the pool after container termination.

The "init" process is the first to run in a SOCK container; init creates a set of new namespaces with a call to `unshare`. The arguments to the call indicate that mount and network namespaces should not be used, because these were the two namespaces that scale poorly (§2.1 and §2.2). Mount namespaces are unnecessary because SOCK uses `chroot`. Network namespaces are unnecessary because requests arrive over Unix domain socket, not over a socket attached to a fixed port number, so port virtualization is not required.

4.2 Generalized Zygotes

Zygote provisioning is a technique where new processes are started as forks of an initial process, the Zygote,

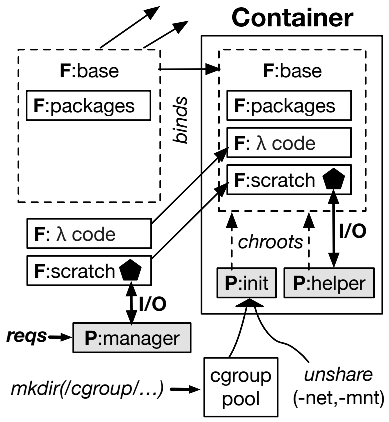


Figure 10. Lean Containers

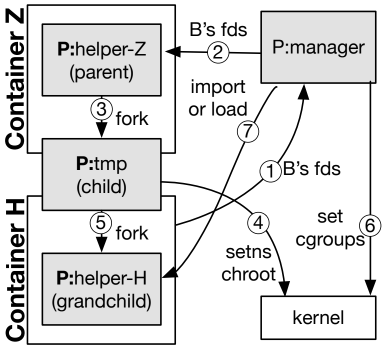


Figure 11. Generalized Zygotes

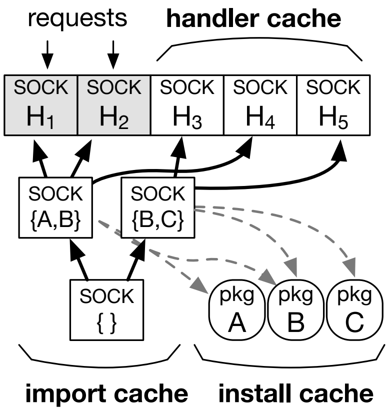


Figure 12. Serverless Caching

that has already pre-imported various libraries likely to be needed by applications, thereby saving child processes from repeatedly doing the same initialization work and consuming excess memory with multiple identical copies. Zygotes were first introduced on Android systems for Java applications [8]. We implement a more general Zygote-provisioning strategy for SOCK. Specifically, SOCK Zygotes differ as follows: (1) the set of pre-imported packages is determined at runtime based on usage, (2) SOCK scales to very large package sets by maintaining multiple Zygotes with different pre-imported packages, (3) provisioning is fully integrated with containers, and (4) processes are not vulnerable to malicious packages they did not import.

As already described, SOCK containers start with two processes, an init process (responsible for setting up namespaces) and a helper process. The helper process is a Python program that listens on the SOCK communication channel; it is capable of (a) pre-importing modules and (b) loading lambda handlers to receive subsequent forwarded events. These two capabilities are the basis for a simple Zygote mechanism. A *Zygote helper* first pre-imports a set of modules. Then, when a lambda is invoked requiring those modules, the Zygote helper is forked to quickly create a new *handler helper*, which then loads the lambda code to handle a forwarded request.

We assume packages that may be pre-imported may be malicious [48], and handlers certainly may be malicious, so both Zygote helpers and handler helpers must run in containers. The key challenge is using Linux APIs such that the forked process lands in a new container, distinct from the container housing the Zygote helper.

Figure 11 illustrates how the SOCK protocol provisions a helper handler ("helper-H" in "Container H") from a helper Zygote ("helper-Z" in "Container Z"). **(1)** The manager obtains references, represented as file descriptors (fds), to the namespaces and the root file system of the new container. **(2)** The fds are passed to helper-Z, which **(3)** forks a child process, "tmp". **(4)** The child then changes roots to the new container with a combination of `fchdir(fd)` and `chroot(".")` calls. The child also calls `setns` (set namespace) for each namespace to relocate to the new container. **(5)** One peculiarity of `setns` is that after the call, the relocation has only partially been applied to all namespaces for the caller. Thus, the child calls fork again, creating a grandchild helper ("helper-H" in the figure) that executes fully in the new container with respect to namespaces. **(6)** The manager then moves the grandchild to the new cgroup. **(7)** Finally, the helper listens on the channel for the next commands; the manager will direct the helper to load the lambda code, and will then forward a request to the lambda.

The above protocol describes how SOCK provisions a handler container from a Zygote container. When OpenLambda starts, a single Zygote that imports no modules is always provisioned. In order to benefit from preimporting modules, SOCK can create additional Zygotes that import various module subsets. Except for the first Zygote, new Zygotes are provisioned from existing Zygotes. The protocol for provisioning a new Zygote container is identical to the protocol for provisioning a new handler container, except for the final step 7. Instead of loading handler code and processing requests, a new Zygote pre-imports a specified list of modules, then waits to be used for the provisioning of other containers.

Provisioning handlers from Zygotes and creating new Zygotes from other Zygotes means that all the interpreters form a tree, with copy-on-write memory unbroken by any call to `exec`. This sharing of physical pages between processes reduces memory consumption [2]. Initialization of the Python runtime and packages will only be done once, and subsequent initialization will be faster.

If a module loaded by a Zygote is malicious, it may interfere with the provisioning protocol (*e.g.*, by modifying the helper protocol so that calls to `setns` are skipped). Fortunately, the Zygote is sandboxed in a container, and

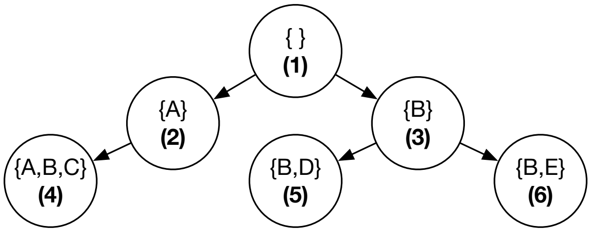


Figure 13. Tree Cache. *Numbered circles represent Zygotes in the cache, and sets of letters indicate the packages imported by a process. Arrows represent the parent-child relationships between interpreter processes.*

will never be passed descriptors referring to unrelated containers, so a malicious process cannot escape into arbitrary containers or the host. SOCK protects innocent lambdas by never initializing them from a Zygote that has pre-imported modules not required by the lambda.

4.3 Serverless Caching

We use SOCK to build a three-tier caching system, shown in Figure 12. First, a *handler cache* maintains idle handler containers in a paused state; the same approach is taken by AWS Lambda [49]. Paused containers cannot consume CPU, and unpausing is faster than creating a new container; however, paused containers consume memory, so SOCK limits total consumption by evicting paused containers from the handler cache on an LRU basis.

Second, an *install cache* contains a large, static set of pre-installed packages on disk. Our measurements show that 97% of installable PyPI packages could coexist in such a installation. This installation is mapped read-only into every container for safety. Some of the packages may be malicious, but they do no harm unless a handler chooses to import them.

Third, an *import cache* is used to manage Zygotes. We have already described a general mechanism for creating many Zygote containers, with varying sets of packages pre-imported (§4.2). However, Zygotes consume memory, and package popularity may shift over time, so SOCK decides the set of Zygotes available based on the import-cache policy. Import caching entails new decisions for handling hits. In traditional caches, lookup results in a simple hit or miss; in contrast, SOCK always hits at least one cache entry and often must decide between alternative Zygotes. Eviction is also complicated by copy-on-write sharing of memory pages between Zygotes, which obfuscates the consumption of individuals. We now describe SOCK's selection and eviction policies.

Import-Cache Selection: Suppose (in the context of Figure 13) that a handler is invoked that requires packages A and B. Entry 4 is a tempting choice to use as the template for our new interpreter; it would provide the best performance because all requisite packages are already imported. However, if package C is malicious, we expose the handler to code that it did not voluntarily import. We could potentially vet a subset of packages to be deemed safe, but we should generally not use cache entries that pre-import packages not requested by a handler. This leaves cache Entries 2 and 3 as reasonable candidates. The import cache decides between such alternatives by choosing the entry with the most matching packages, breaking ties randomly. When SOCK must use an entry X that is not an exact match, it first replicates X to a new entry Y, imports the remaining packages in Y, and finally replicates from Y to provision for the handler.

Import-Cache Eviction: The import cache measures the cumulative memory utilization of all entries; when utilization surpasses a limit, a background process begins evicting entries. Deciding which interpreters to evict is challenging because the shared memory between interpreters makes it difficult to account for the memory used by a particular entry. The import cache relies on a simple runtime model to estimate potential memory reclamation; the model identifies the packages included by an interpreter that are not included by the parent entry. The model uses the on-disk size of the packages as a heuristic for estimating memory cost. The import cache treats the sum of these sizes as the *benefit* of eviction and the number of uses over a recent time interval as the *cost* of eviction, evicting the entry with highest benefit-to-cost ratio.

5. Evaluation

We now evaluate the performance of SOCK relative to Docker-based OpenLambda and other platforms. We run experiments on two m510 machines [11] with the 4.13.0-37 Linux kernel: a package mirror and an OpenLambda worker. The machines have 8-core 2.0 GHz Xeon D-1548 processors, 64 GB of RAM, and a 256 GB NVMe SSD. We allocate 5 GB of memory for the handler cache and 25 GB for the import cache. We consider the following questions: *What speedups do SOCK containers provide OpenLambda* (§5.1)? *Does built-in package support reduce cold-start latency for applications with dependencies* (§5.2)? *How does SOCK scale with the number of lambdas and packages* (§5.3)? And *how does SOCK compare to other platforms for a real workload* (§5.4)?

5.1 Container Optimizations

SOCK avoids many of the expensive operations necessary to construct a general-purpose container (*e.g.*, network namespaces, layered file systems, and fresh cgroups). In order to evaluate the benefit of lean containerization, we concurrently invoke no-op lambdas on OpenLambda, using either Docker or SOCK as the container engine. We disable all SOCK caches and Zygote preinitialization. Figure 14 shows the request throughput and average latency as we vary the number of concurrent outstanding requests. SOCK is strictly faster on both metrics, regardless of concurrency. For 10 concurrent re-

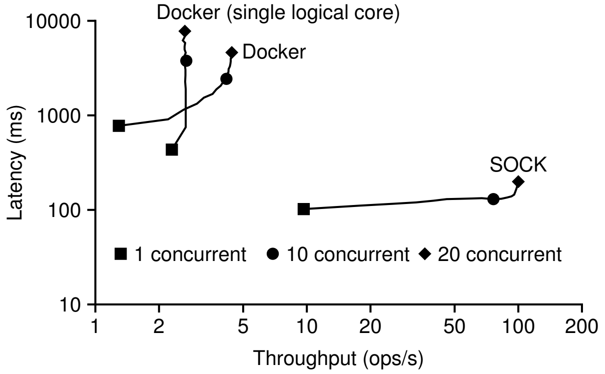


Figure 14. Docker vs. SOCK. *Request throughput (x-axis) and latency (y-axis) are shown for SOCK (without Zygotes) and Docker for varying concurrency.*

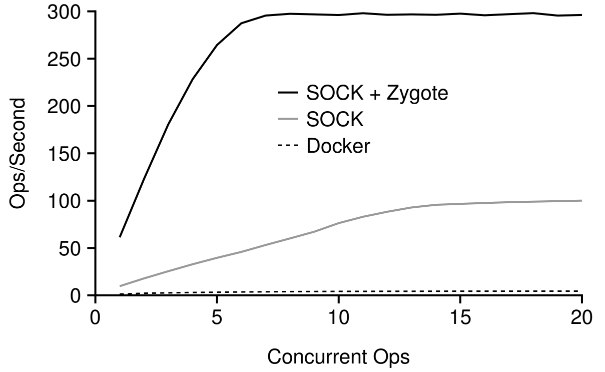


Figure 15. Interpreter Preinitialization. *HTTP Request Throughput is shown relative to number of concurrent requests.*

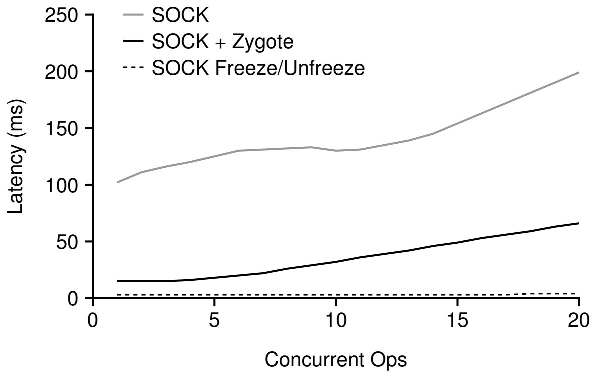


Figure 16. Container Reuse vs. Fast Creation. *SOCK container creation is compared to the freeze/unfreeze operation that occurs when there are repeated calls to the same lambda.*

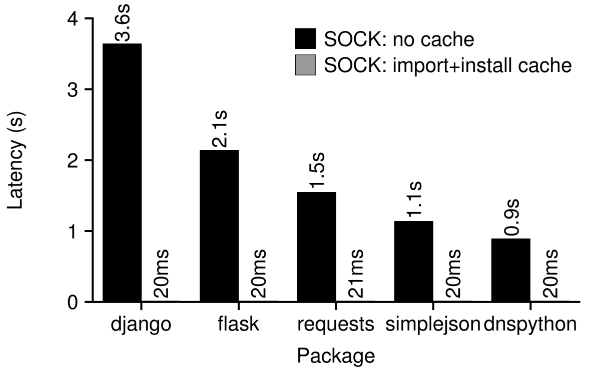


Figure 17. Pre-Imported Packages. *SOCK latency with and without package caches are shown.*

quests, SOCK has a throughput of 76 requests/second (18× faster than Docker) with an average latency of 130 milliseconds (19× faster). Some of the namespaces used by Docker rely heavily on RCUs (§2.2), which scale poorly with the number of cores [34]. Figure 14 also shows Docker performance with only one logical core enabled: relative to using all cores, this reduces latency by 44% for $concurrency = 1$, but throughput no longer scales with concurrency.

SOCK also improves performance by using Zygote-style preinitialization. Even if a lambda uses no libraries, provisioning a runtime by forking an existing Python interpreter is faster than starting from scratch. Figure 15 compares SOCK throughput with and without Zygote preinitialization. Using Zygotes provides SOCK with an additional 3× throughput improvement at scale.

OpenLambda, like AWS Lambda [49], keeps recently used handlers that are idle in a paused state in order to avoid cold start should another request arrive. We now compare the latency of SOCK cold start to the latency of unpause, as shown in Figure 16. Although Zygotes have reduced no-op cold-start latency to 32 ms ($concurrency = 10$), unpausing takes only 3 ms. Although SOCK cold-start optimizations enable more aggressive resource reclamation, it is still beneficial to pause idle handlers before immediately evicting them.

5.2 Package Optimizations

SOCK provides two package-oriented optimizations. First, SOCK generalizes the Zygote approach so that new containers can be allocated by one of many different Zygote containers, each with different packages pre-imported, based on the current workload (import caching). Second, a large subset of packages are pre-installed to a partition that is bind-mounted read-only in every container (install caching).

We first evaluate these optimizations together with a simple workload, where a single task sequentially invokes different lambdas that use the same single library, but perform no work. Figure 17 shows the result. Without optimizations, downloading, installing, and importing usually takes at least a second. The optimizations reduce latency to 20 ms, at least a 45× improvement.

To better understand the contributions of the three caching layers (*i.e.*, the new import and install caches and the old handler cache), we repeat the experiment in Figure 17 for django, evaluating all caches, no caches, and

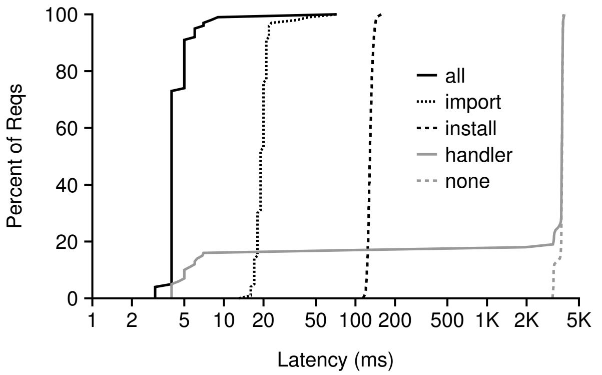


Figure 18. Individual Caches. *Each line shows a latency CDF for a different configuration for the django experiment.*

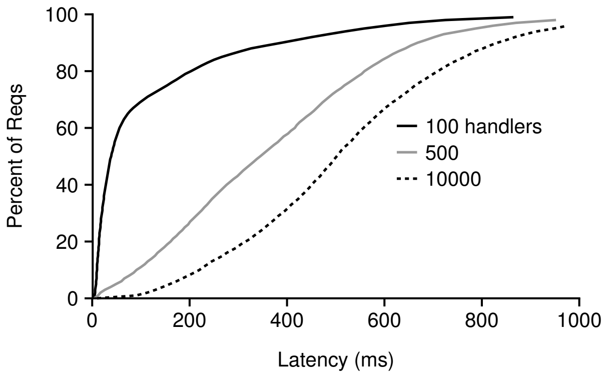


Figure 20. Scalability: Synthetic Packages. *Each line represents a CDF of latencies for a different working-set size.*

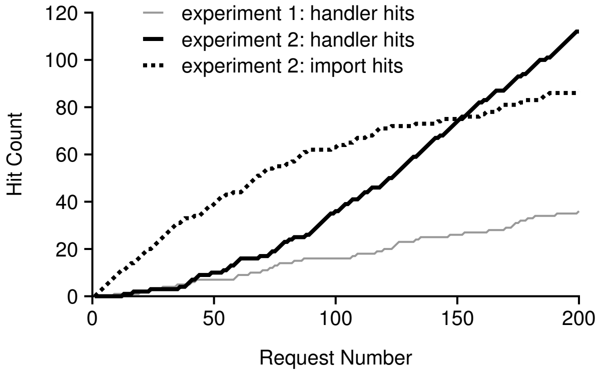


Figure 19. Handler/Import Cache Interactions. *Hit counts are shown during cache warmup (the first 200 requests) for two cache configurations.*

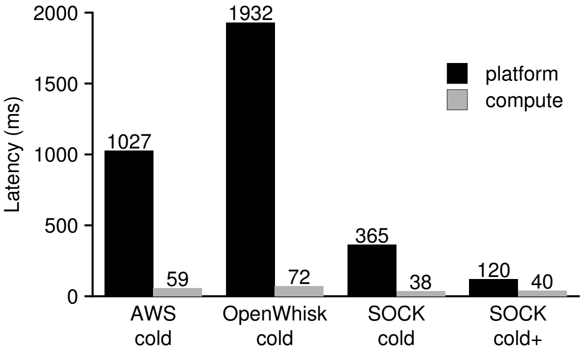


Figure 21. AWS Lambda and OpenWhisk. *Platform and compute costs are shown for cold requests to an image-resizing lambda. S3 latencies are excluded to minimize noise.*

each cache in isolation. For each experiment, 100 different lambdas import django, and a single task sequentially invokes randomly-chosen lambdas. Figure 18 shows the results. The handler cache has bimodal latency: it usually misses, but is fastest upon a hit. The working set fits in the import cache, which provides consistent latencies around 20 ms; the install cache is also consistent, but slower. Using all caches together provides better performance than any one individually.

When import caching is enabled, processes in the handler cache and processes in the import cache are part of the same process tree. This structure leads to deduplication: multiple processes in the handler cache can share the same memory page on a copy-on-write basis with a parent process in the import cache. This allows the handler cache to maintain more cache entries. Figure 19 illustrates this helpful interaction. We issue 200 requests to many different lambdas, all of which import django, without an import cache (experiment 1) and with an import cache (experiment 2). In the first experiment, the handler cache has 18% hits. In the second, deduplication allows the handler cache to maintain more entries, achieving 56% hits.

5.3 Scalability

We stress test SOCK with a large set of artificial packages (100K). The packages generate CPU load and memory load, similar to measured overheads of 20 popular packages (§3.1). We create dependencies between packages similar to the PyPI dependency structure. Each handler imports 1-3 packages directly. The packages used are decided randomly based on package popularity; popularity is randomly assigned with a Zipfian distribution, $s = 2.5$. All packages are pre-installed to the install cache.

We also vary the number of handlers (100 to 10K). A small working set exercises the handler cache, and a large working set exercises the install cache. The import cache should service mid-sized working sets. Handlers are executed uniformly at random as fast as possible by 10 concurrent tasks. Figure 20 shows a latency CDF for each working set size. With 100 handlers, SOCK achieves low latency (39 ms median). For 10K handlers, 88% percent of requests must be serviced from the install cache, so the median latency is 502 ms. For 500 handlers, the import cache absorbs 46% of the load, and the handler cache absorbs 6.4%, resulting in 345 ms latencies.

5.4 Case Study: Image Resizing

In order to evaluate a real serverless application, we implement on-demand image resizing [28]. A lambda reads an image from AWS S3, uses the Pillow package to resize it [10], and writes the output back to AWS S3. For this experiment, we compare SOCK to AWS Lambda and OpenWhisk, using 1 GB lambdas (for AWS Lambda) and a pair of m4.xlarge AWS EC2 instances (for SOCK and OpenWhisk); one instance services requests and the other hosts handler code. We use AWS's US East region for EC2, Lambda, and S3.

For SOCK, we preinstall Pillow and the AWS SDK [44] (for S3 access) to the install cache and specify these as handler dependencies. For AWS Lambda and OpenWhisk, we bundle these dependencies with the handler itself, inflating the handler size from 4 KB to 8.3 MB. For each platform, we exercise cold-start performance by measuring request latency after re-uploading our code as a new handler. We instrument handler code to separate compute and S3 latencies from platform latency.

The first three bars of Figure 21 show compute and platform results for each platform (average of 50 runs). "SOCK cold" has a platform latency of 365 ms, 2.8× faster than AWS Lambda and 5.3× faster than OpenWhisk. "SOCK cold" compute time is also shorter than the other compute times because all package initialization happens after the handler starts running for the other platforms, but SOCK performs package initialization work as part of the platform. The "SOCK cold+" represents a scenario similar to "SOCK cold" where the handler is being run for the first time, but a different handler that also uses the Pillow package has recently run. This scenario further reduces SOCK platform latency by 3× to 120 ms.

6. Related Work

Since the introduction of AWS Lambda in 2014 [5], many new serverless platforms have become available [6, 14, 17, 39, 45]. We build SOCK over OpenLambda [20]. SOCK implements and extends our earlier Pipsqueak proposal for efficient package initialization [38].

In this work, we benchmark various task-provisioning primitives and measure package initialization costs. Prior studies have ported various applications to the lambda model in order to evaluate platform performance [21, 33]. Spillner *et al.* [46] ported Java applications to AWS Lambda to compare performance against other platforms, and Fouladi *et al.* [16] built a video encoding platform over lambdas. Wang *et al.* [50] reverse engineer many design decisions made by major serverless platforms.

There has been a recent revival of interest in sandboxing technologies. Linux containers, made popular through Docker [36], represent the most mainstream technology; as we show herein, the state of the art is not yet tuned to support lambda workloads. OpenWhisk, which uses Docker containers, hides latency by maintaining pools of ready containers [47]. Recent alternatives to traditional containerization are based on library operating systems, enclaves, and unikernels [7, 24, 29, 31, 41, 42].

The SOCK import cache is a generalization of the Zygote approach first used by Android [8] for Java processes. Akkus *et al.* [1] also leverage this technique to efficiently launch multiple lambdas in the same container when the lambdas belong to the same application. Zygotes have also been used for browser processes (sometimes in conjunction with namespaces [12]). We believe SOCK's generalized Zygote strategy should be generally applicable to other language runtimes that dynamically load libraries or have other initialization costs such as JIT-compilation (*e.g.*, the v8 engine for Node.js [25] or the CLR runtime for C# [3, 30]); however, it is not obvious how SOCK techniques could be applied to statically-linked applications (*e.g.*, most Go programs [53]).

Process caching often has security implications. For example, HotTub [27] reuses Java interpreters, but not between different Linux users. Although the Zygote approach allows cache sharing between users, Lee *et al.* [26] observed that forking many child processes from the same parent without calling `exec` undermines address-space randomization; their solution was Morula, a system that runs `exec` every time, but maintains a pool of preinitialized interpreters; this approach trades overall system throughput for randomization.

7. Conclusion

Serverless platforms promise cost savings and extreme elasticity to developers. Unfortunately, these platforms also make initialization slower and more frequent, so many applications and microservices may experience slowdowns if ported to the lambda model. In this work, we identify container initialization and package dependencies as common causes of slow lambda startup. Based on our analysis, we build SOCK, a streamlined container system optimized for serverless workloads that avoids major kernel bottlenecks. We further generalize Zygote provisioning and build a package-aware caching system. Our hope is that this work, alongside other efforts to minimize startup costs, will make serverless deployment viable for an ever-growing class of applications.

Acknowledgements

We thank the anonymous ATC reviewers, the members of ADSL, and our colleagues at GSL for their valuable input. This material was supported by funding from NSF grant CNS-1421033, DOE grant DESC0014935, and student funding from Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, Microsoft, or other institutions.

References

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018.

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[3] Assemblies in the Common Language Runtime. https://docs.microsoft.com/en-us/dotnet/framework/app-domains/assemblies-in-the-common-language-runtime, May 2018.

[4] AWS Developer Forums: Java Lambda Inappropriate for Quick Calls? https://forums.aws.amazon.com/thread.jspa?messageID=679050, July 2015.

[5] AWS Lambda. https://aws.amazon.com/lambda/, February 2018.

[6] Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/functions/, February 2018.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[8] Dan Bornstein. Dalvik Virtual Machine Internals. Talk at Google I/O, 2008.

[9] cgroups - Linux Control Groups. http://man7.org/linux/man-pages/man7/cgroups.7.html, May 2018.

[10] Alex Clark. Pillow Python Imaging Library. https://pillow.readthedocs.io/en/latest/, February 2018.

[11] CloudLab. https://www.cloudlab.us/, February 2018.

[12] Chromium Docs. Linux Sandboxing. https://chromium.googlesource.com/chromium/src/+/lkcr/docs/linux_sandboxing.md, February 2018.

[13] Eric Dumazet. Re: net: cleanup_net is slow. https://lkml.org/lkml/2017/4/21/533, April 2017.

[14] Alex Ellis. Introducing Functions as a Service (OpenFaaS). https://blog.alexellis.io/introducing-functions-as-a-service/, August 2017.

[15] Jeremy Elson and Jon Howell. Handling Flash Crowds from Your Garage. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 171–184. USENIX Association, 2008.

[16] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.

[17] Google Cloud Functions. https://cloud.google.com/functions/docs/, February 2018.

[18] Jim Gray. Why Do Computers Stop and What Can We Do About It? In *6th International Conference on Reliability and Distributed Databases*, June 1987.

[19] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 71–83. ACM, 2011.

[20] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.

[21] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

[22] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 293–304. ACM, 2002.

[23] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 17–17. USENIX Association, 2012.

[24] Nicolas Lacasse. Open-Sourcing gVisor, a Sandboxed Container Runtime. https://cloudplatform.googleblog.com/2018/05/Open-sourcing-gVisor-a-sandboxed-container-runtime.html, May 2018.

[25] Thibault Laurens. How the V8 Engine Works? http://thibaultlaurens.github.io/javascript/2013/04/29/how-the-v8-engine-works/, April 2013.

[26] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From Zygote to Morula: Fortifying Weakened ASLR on Android. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 424–439. IEEE, 2014.

[27] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't Get Caught In the Cold, Warm-up Your JVM. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, Georgia, October 2016.

[28] Bryan Liston. Resize Images on the Fly with Amazon S3, AWS Lambda, and Amazon API Gateway. https://aws.amazon.com/blogs/compute/resize-images-on-the-fly-with-amazon-s3-aws-lambda-and-amazon-api-gateway/, January 2017.

[29] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Uniker-

nels: Library Operating Systems for the Cloud. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, Houston, Texas, March 2013.

[30] Managed Execution Process. https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process, May 2018.

[31] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233. ACM, 2017.

[32] Linux Manpages. pivot_root (2) - Linux Man Pages. https://www.systutorials.com/docs/linux/man/2-pivot_root/, February 2018.

[33] Garrett McGrath and Paul R. Brenner. Serverless Computing: Design, Implementation, and Performance. In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*, pages 405–410. IEEE, 2017.

[34] Paul E McKenney. Introduction to RCU Concepts: Liberal application of procrastination for accommodation of the laws of physics for more than two decades! In *LinuxCon Europe 2013*, October 2013.

[35] Paul E McKenney, Silas Boyd-Wickizer, and Jonathan Walpole. RCU Usage in the Linux Kernel: One Decade Later, 2013.

[36] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal, Issue 239, March 2014.

[37] namespaces(7) - overview of Linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html, May 2018.

[38] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Pipsqueak: Lean Lambdas with Large Libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 395–400, June 2017.

[39] IBM Cloud Functions. https://www.ibm.com/cloud/functions, February 2018.

[40] Rob Pike. Another Go at Language Design. http://www.stanford.edu/class/ee380/Abstracts/100428.html, April 2010.

[41] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304. ACM, 2011.

[42] Nelly Porter, Jason Garms, and Sergey Simakov. Introducing Asylo: an Open-Source Framework for Confidential Computing. https://cloudplatform.googleblog.com/2018/05/Introducing-Asylo-an-open-source-framework-for-confidential-computing.html, May 2018.

[43] Eric Ries. *The Lean Startup*. Crown Business, September 2011.

[44] Amazon Web Services. The AWS SDK for Python. https://boto3.readthedocs.io/en/latest/, February 2018.

[45] Shaun Smith. Announcing Fn: An Open Source Serverless Functions Platform. https://blogs.oracle.com/developers/announcing-fn, October 2017.

[46] Josef Spillner and Serhii Dorodko. Java Code Analysis and Transformation into AWS Lambda Functions. *CoRR*, abs/1702.05510, 2017.

[47] Markus Thömmes. Squeezing the milliseconds: How to make serverless platforms blazing fast! https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0, April 2017.

[48] Nikolai Philipp Tschacher. Typosquatting in Programming Language Package Managers. Bachelor Thesis, University of Hamburg (6632193). http://incolumitas.com/data/thesis.pdf, March 2016.

[49] Tim Wagner. Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, December 2014.

[50] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018.

[51] Matt Weinberger. The 15 Most Popular Programming Languages, According to the 'Facebook for Programmers'. http://www.businessinsider.com/the-9-most-popular-programming-languages-according-to-the-facebook-for-programmers-2017-10#1-javascript-15, October 2017.

[52] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration Debugging As Search: Finding the Needle in the Haystack. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 6–6. USENIX Association, 2004.

[53] Matt Williams. Go executables are statically linked, except when they are not. http://matthewkwilliams.com/index.php/2014/09/28/go-executables-are-statically-linked-except-when-they-are-not/, September 2014.

[54] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. A Smart Hill-Climbing Algorithm for Application Server Configuration. In *Proceedings of the 13th international conference on World Wide Web*, pages 287–296. ACM, 2004.

DynaMix: Dynamic Mobile Device Integration for Efficient Cross-device Resource Sharing

Dongju Chae[¶] Joonsung Kim[†] Gwangmu Lee[†]
Hanjun Kim[¶] Kyung-Ah Chang[*] Hyogun Lee[*] Jangwoo Kim[†]
[¶]*POSTECH* [†]*Seoul National University* [*]*Samsung Electronics*

Abstract

In the era of the Internet of Things, users desire more valuable services by simultaneously utilizing various resources available in remote devices. As a result, cross-device resource sharing, a capability to utilize the resources of a remote device, becomes a desirable feature to enable interesting multi-device services. However, the existing resource sharing mechanisms either have limited resource coverage, involve complex programming efforts for utilizing multiple devices, or more importantly, incur huge inter-device network traffic.

We propose DynaMix, a novel framework that realizes efficient cross-device resource sharing. First, DynaMix maximizes resource coverage by dynamically integrating computation and I/O resources of remote devices with distributed shared memory and I/O request forwarding. Second, DynaMix obviates the need for multi-device programming by providing the resource sharing capability at the low level. Third, DynaMix minimizes inter-device network traffic by adaptively redistributing tasks between devices based on their dynamic resource usage. By doing so, DynaMix achieves efficient resource sharing along with dynamic plug-and-play and reconfigurability. Our example implementation on top of Android and Tizen devices shows that DynaMix enables efficient cross-device resource sharing in multi-device services.

1 Introduction

In the era of the Internet of Things, a user can access an increasing number of heterogeneous devices (e.g., smartphones, wearable devices, smart TVs) equipped with diverse, and possibly different, hardware resources (e.g., CPU, memory, camera, screen). As a result, such an environment poses the need for multi-device services which simultaneously utilize the diverse resources of the heterogeneous devices. For instance, when watching movies or viewing PDF files, a user can use a large TV screen rather than a smaller smartphone screen. Also, a user can take pictures from various angles by using multiple remote cameras. In a similar sense, a number of recent studies [33,37,38] develop and demonstrate multi-device services utilizing resources of multiple devices.

However, the existing cross-device resource sharing schemes suffer from several challenges. First, using network libraries explicitly imposes significant programming burden on developers [2, 3, 6] as they should follow a server-client model that involves careful task distribution between server and client processes. Distributed programming platform [54] may reduce the programming burden; however, they still impose the burden of efficiently partitioning an application. Second, code offloading [19, 21, 28] and remote I/O [11] can enable cross-device resource sharing without the programming burden. Unfortunately, neither of them supports all computation (e.g., CPU, memory) and I/O sharing at the same time, which limits their applicability. More importantly, the existing schemes do not optimize the placement of tasks and hence suffer when running on slow wireless networks.

Motivated by the limitations of the existing mechanisms, we need a new cross-device resource sharing mechanism achieving all of the following design goals. First, it should fully integrate the diverse resources of different devices including CPU, memory, and I/O resources. Second, it should achieve good programmability by not exposing any cross-device resource sharing details to the application layer. Third, it should dynamically redistribute tasks between devices to minimize the negative performance impacts of slow wireless networks.

In this paper, we propose **DynaMix**, a novel framework to enable **Dyna**mic **M**obile device **i**ntegration for efficient **cross**-device resource sharing. First, DynaMix fully integrates diverse resources using Distributed Shared Memory (DSM) and I/O request forwarding; DSM integrates CPU and memory, and I/O request forwarding integrates I/O resources. Second, as DSM and I/O request forwarding enable low-level resource sharing below the application level, DynaMix does not demand applications to be aware of multiple devices, achieving good programmability. Third, DynaMix dynamically redistributes tasks between devices in a way that minimizes inter-device communication by monitoring per-device resource usage and inter-device network usage. In addition, DynaMix supports seamless plug-and-play of remote devices by monitoring their connectivity and by taking checkpoints of an application's states.

For evaluation, we implement DynaMix on various Android and Tizen devices (e.g., Google Nexus, Samsung Smart TV). We also introduce three multi-device services to demonstrate the effectiveness of DynaMix: home theater, smart surveillance, and photo classification. The experimental results clearly show that DynaMix enables efficient cross-device resource sharing by fully integrating diverse resources and by dynamically redistributing tasks between devices. For instance, DynaMix achieves the target performance goal of home theater (i.e., 24 FPS when playing HD movies), whereas the existing mechanisms suffer from severe performance degradation (e.g., only 8.2 FPS with request forwarding).

In summary, our contributions are as follows:

- **Novel Platform.** We propose DynaMix, a novel framework to fully integrate remote resources for efficient cross-device resource sharing.
- **High Applicability.** DynaMix can easily be deployed to existing devices, and its low-level resource sharing enables easy programmability.
- **High Performance.** DynaMix minimizes the inter-device communication overheads by dynamically redistributing tasks between devices.
- **High Reliability.** DynaMix supports seamless plug-and-play of remote devices, improving the reliability of multi-device services.

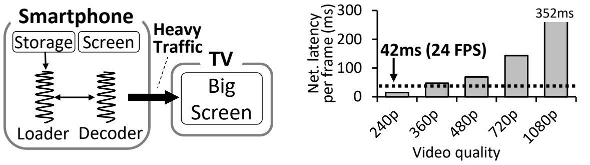
2 Background and Motivation

In the IoT environment, cross-device resource sharing is a promising solution to satisfy various service demands of users who can access an increasing number of heterogenous devices. The users can select favorable resources in different devices, so that they enjoy the same application in different ways depending on their resource configurations.

2.1 Limitations of Existing Schemes

To enable multi-device services, researchers have proposed various resource sharing schemes. We group them into three categories and compare their tradeoffs.

I/O Request Forwarding. The I/O request forwarding is a method to utilize remote I/O resources (e.g., camera, screen, audio, sensor) by forwarding I/O requests to the target device, which then accesses the requested resources on behalf of the requesting device. The request forwarding schemes can forward the I/O requests in different layers (e.g., kernel, platform, user). For example, Rio [11] forwards I/O requests at the kernel level to a remote device which then performs the delivered I/O requests. M+ [43] provides cross-device functionality sharing at the platform level by forwarding IPC messages. Both schemes enable the transparent access to remote I/O resources. On the other hand, user-level [2,3,6] request forwarding schemes make programmers explicitly handle the remote I/O requests.



(a) An example setup

(b) Network latency

Figure 1: An example setup to play a video on a remote screen and network latency to send a single frame

However, the applicability of the existing request forwarding schemes is limited as follows. First, they support only I/O resources for resource sharing[1]. Next, they require carefully-designed abstraction layers to support single-device applications. Furthermore, they can suffer from severe network overheads unless they access resources in an optimized task distribution. Figure 1a shows an example kernel-level request forwarding setup configured to use a remote screen to play a video. Since the local device forwards the decoded frame to the remote screen, it can suffer from the severe communication overhead as the video quality increases. Figure 1b shows that only the lowest resolution quality can barely meet the 24 frames per second (FPS) performance goal. Actually, moving `Decoder` task from the smartphone to the TV would greatly reduce the network overheads as only the small traffic between `Loader` and `Decoder` is exposed. From this example, we can see why the resource-aware task redistribution is important.

Code Offloading and Distributed Computation. The code offloading [19, 21] and distributed computation [28] schemes utilize remote computation resources (e.g., CPU, memory) by offloading performance-critical code regions to more powerful devices. They can not only improve the performance but also save the power consumption of the requesting device by using a faster CPU or exploiting the increased parallelism with more cores. In addition, COMET [28] implements a software-based distributed shared-memory (DSM) framework to support efficient thread offloading among devices.

However, the applicability of the existing code of-

[1] Note that M+ [43] restrictively uses CPU and memory resources for specific platform services.

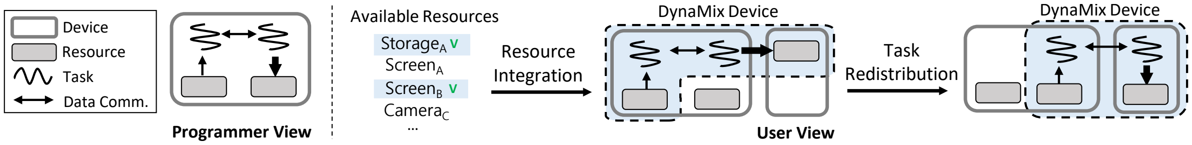


Figure 2: An example workflow of the DynaMix framework. Though programmers develop single-device applications, users can configure their services (i.e., DynaMix devices) by selecting desired resources to access remote resources.

floading schemes is limited as follows. First, they support only computation resources for cross-device sharing, which leaves I/O resources to be wasted. Next, their migration points of non-DSM schemes are restricted to specific function entries, similar to Remote Procedure Calls (RPC). Also, the migrated tasks should eventually go back to the requesting device which restricts the scope of performance-critical task redistributions.

Distributed Programming Platform. A distributed programming platform such as Sapphire [54] is similar to the distributed computation scheme but provides an interface to enable more flexible task migrations. Once the tasks are deployed by the platform-defined unit objects, the platform supports a limited form of task redistributions to reduce the performance overhead.

However, such distributed programming platform suffers from the following limitations. First of all, the resource coverage is still limited to computation resources for cross-device sharing. Next, the scheme leaves the burden of difficult multi-device programming (e.g., device-aware task partitioning, dynamic exception handling) to application developers.

2.2 Design Goals

Motivated by the limitations, we claim that an ideal resource sharing framework must satisfy the following.

High Resource Coverage. The framework should cover both I/O and computation resources for cross-device sharing. Various types of I/O resources enable the framework to provide interesting multi-device services which are infeasible in a single device alone due to its limited capabilities (e.g., device's unsupported resource types and physical location). In addition, sharing computation resources allows an application to run in a more efficient way by distributing its tasks across other devices.

Single-device Application Support. The framework should transparently support single-device applications for multi-device services. Developing multi-device applications [2, 3, 6] using a server-client model often imposes an excessive burden on developers (e.g., statically separated multiple programs). Also, this approach is practically limited toward satisfying users' various demands and developers have to manually handle dynamic behaviors. On the other hand, if the framework transparently supports a single-device application to access remote resources, developers no longer consider how remote resources are accessed. Users create their own service by selecting favorable resources and the framework provides seamless mechanisms to access them, significantly reducing the programming burden.

Resource-aware Task Redistribution. The framework should minimize the inter-device communication overhead with dynamic inter-device task redistributions. The communication overhead incurred by the remote access highly depends on dynamic factors, such as the reconfiguration of the resource sharing, the available network bandwidth, and runtime behaviors of tasks in an application. Therefore, it is important to adaptively redistribute tasks to the optimal devices to minimize the overhead.

3 DynaMix Framework

3.1 Overview

Figure 2 shows an example workflow of DynaMix framework. First, programmers develop DynaMix applications. To reduce the burden of the programmers, DynaMix requires neither any special programming concepts nor special APIs except the underlying memory consistency model described in §4.1. Therefore, programmers can write ordinary multi-threaded programs on a single device with multi-thread libraries without concerns about remote resources. This single-device programming model of DynaMix makes developing new applications and porting existing applications easy. Second, users can select desired resources (e.g., Storage_A and Screen_B in Figure 2) to execute DynaMix applications at runtime. DynaMix framework dynamically integrates the selected resources and constructs a single virtual device called a *DynaMix device*. Third, DynaMix detects the network traffic and automatically redistributes tasks across the devices to minimize the network overhead. Within the DynaMix device, tasks (i.e., threads) of the DynaMix applications can freely access remote resources or be migrated for the optimal task redistribution.

3.2 DynaMix Operations

DynaMix framework has two basic operation models: *remote resource integration* and *resource-aware task redistribution*. To support the operations, users should first make their devices DynaMix-enabled by installing two software components on each device: *resource integrator* and *thread migrator*. The resource integrator integrates both computation and I/O resources (or constructs a DynaMix device) by applying a distributed shared

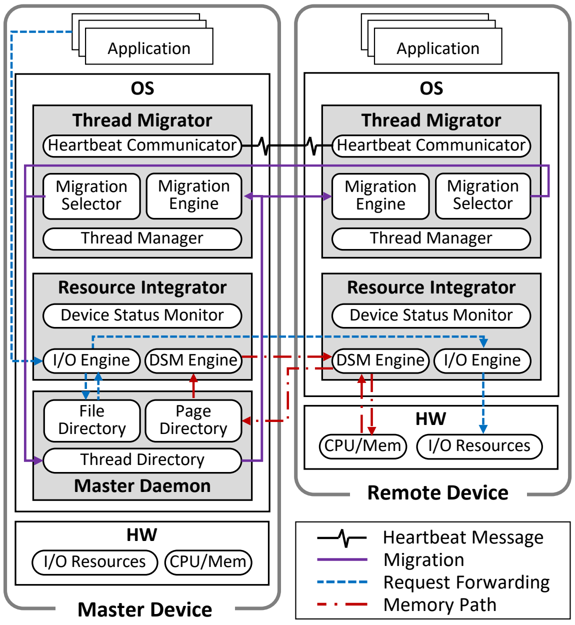


Figure 3: DynaMix architectural overview

memory (DSM) model and I/O request forwarding together (§3.2.1). The thread migrator monitors both inter-thread communication and device connectivity, and dynamically redistributes threads to their optimal locations to minimize the communication overhead (§3.2.2).

3.2.1 Remote Resource Integration

The resource integrators installed on each device collaboratively apply a DSM model and a kernel-level I/O request forwarding to integrate both computation (e.g., CPU, memory) and I/O (e.g., display, storage) resources. This mechanism enables DynaMix to satisfy two design goals of ideal resource sharing: **single-device programming model** and **high resource coverage**.

The resource integrator performs the integration in three steps. First, the resource integrator collects the information of remote resources (e.g., CPU frequency, memory size, I/O type), broadcasted by remote resource integrators, and makes the resources available for user applications. Second, if an application tries to use a remote resource, the resource integrator forwards the request to the target resource integrator. Third, the target resource integrator delivers the outcome to the application through the shared memory for computation results or through forwarding for I/O results.

3.2.2 Resource-aware Task Redistribution

With only the I/O request forwarding, DynaMix can incur severe inter-device communications. Therefore, DynaMix applies a resource-aware task redistribution mechanism by adaptively migrating threads to the optimal devices in a way to minimize the overall inter-device traffic. This mechanism satisfies the design goal of **resource-aware task redistributions**.

The resource integrator and the thread migrator work together to enable task redistributions as follows. First, the resource integrator monitors per-thread resource usage (e.g., CPU, network) to detect possible resource contentions. Second, on detecting a contention, the thread migrator compares tradeoffs of various thread allocation scenarios, and finds the best one. Third, the thread migrator migrates threads based on the scenario by delivering their execution contexts to the target devices.

4 Implementation

This section describes how we implement the aforementioned core components (*resource integrator* and *thread migrator*) and a newly introduced *master demon* component to correctly orchestrate operations. The master daemon runs on a failure-free master device on which a user launches applications. Note that we regard failures in the master device as user-intended ones such as device shutdown. Figure 3 illustrates the overall architecture.

4.1 Resource Integrator

The resource integrator consists of three components: a DSM engine, an I/O engine, and a device status monitor. The DSM and I/O engines integrate computation and I/O resources, respectively, and the device status monitor detects intra-device resource contentions.

4.1.1 DSM Engine

The DSM engine integrates the memory regions of multiple devices into a single memory space in a DSM manner. On receiving a memory access request, the DSM engine either delivers its local memory data or forwards a request to the destination DSM engine owning the data. It also works with the master daemon to orchestrate these communications for globally consistent memory management (§4.3.2). The DSM engine applies three performance optimizations as follows. First, it adopts a lazy release consistency (LRC) model [32] to safely delay memory synchronization within acquire-release block, similar to previous work [27,28]. Second, it actively performs memory prefetches on detecting sequential memory access patterns. Third, it uses a page-level coherence block to reduce the coherence overheads.

To support the page-level DSM, the DSM engine leverages a page fault handler in Linux kernel which manages page permissions. When an application enters a critical section (i.e., lock acquire), the DSM engine disables write permissions of all shareable pages in the target application. In this way, the DSM engine can detect the page modifications during a critical section. On the exit of the critical section (i.e., lock release), it recovers the write permissions. Due to the LRC model, the memory transfer of the modified pages occurs only when another device newly acquires the same lock, which minimizes unnecessary network communications.

To further reduce the network traffic, the DSM engine transfers only the updated contents called *diffs*. On a lock release, the DSM engine generates diffs by comparing the contents of original and modified pages. When another device acquires the same lock, its DSM engine receives the corresponding diffs from the previous lock holder and applies them into the original pages.

4.1.2 I/O Engine

The I/O engine manages the access to local and remote I/O resources through kernel-level request forwarding. For the purpose, the I/O engine provides a *device file* boundary for cross-device I/O sharing similar to previous approaches [10, 11]. Mobile platforms with the Linux kernel base (e.g., Android, Tizen) use device files as their I/O abstraction layer because such files are device-agnostic. In this way, DynaMix can support a wide spectrum of I/O resources. To forward incoming requests from the host to a remote target device, the I/O engine intercepts I/O-related system calls (e.g., *open*, *read*, *write*, *ioctl*) and delivers them to the remote device with their input parameters. The remote device then performs the forwarded requests and returns the results to the host. The I/O engine also cooperates with a platform to allow users to access remote I/O resources transparently.

For example, to access audio peripherals (e.g., speaker, microphone) on a remote device, the I/O engine creates virtual device files corresponding to device files for audio peripherals (e.g., `/dev/snd/pcmCxDxx`, `/dev/snd/control`). The host I/O engine transfers requests coming through a virtual device file to the remote I/O engine which executes the requests with the corresponding original device file. Note that an audio Hardware Abstraction Layer (HAL) library (e.g., tinyalsa) is modified to access virtual device files instead of original device files. In this way, DynaMix applications can transparently access the remote audio peripherals.

Unfortunately, such kernel-level request forwarding does not directly support some I/O resources (e.g., a screen, file system) that require special management. For example, to display frame data from a frame buffer (`/dev/graphics/fb0`) in the host, the remote I/O engine should cooperate with graphics APIs in a platform to follow the existing graphics stack (i.e., SurfaceFlinger). In particular, to access a file on a remote storage, the I/O engine works with the master daemon which keeps a file directory containing the file metadata. Therefore, devices joining the DynaMix device should upload their file metadata information to the shared file directory. On receiving a file access request, the I/O engine first checks the local file directory. If the file does not exist, the I/O engine asks the master daemon to find the location in the shared file directory and forwards the request to the owner device.

4.1.3 Device Status Monitor

The device status monitor periodically collects various system information (e.g., per-thread CPU utilization, network stall time) to detect CPU and network contentions. The device status monitor is implemented as a kernel thread, which enables more accurate resource monitoring. It detects CPU contentions when CPUs are fully utilized but each thread has low CPU utilization without the existence of other bottlenecks (e.g., no I/O wait). On the other hands, it detects network contentions when the stalled time due to remote I/O accesses or memory synchronization exceeds a pre-defined threshold [2] (e.g., 30% in our environment). On detecting such contentions, the device status monitor immediately notifies the master daemon to initiate thread redistributions.

4.2 Thread Migrator

The thread migrator consists of four components: a thread manager, a migration selector, a migration engine, and a heartbeat communicator.

4.2.1 Thread Manager

The thread manager preserves various information of running threads such as execution states, resource usage, and locks. On resource contentions, the thread manager calculates threads' data communications[3] (i.e., thread-to-thread and thread-to-resource) and sends the results to the migration selector which determines the best victim for migration and its destination device. The thread manager also implements kernel-level locks to synchronize threads across different devices. Note that we modify a user-level multi-thread library (e.g., POSIX) to access these locks internally. The thread manager checks with the master daemon before allowing a thread to acquire a lock. The master daemon then forces the prior lock holder to transfer the updated memory within the acquire-release block, following the LRC model.

For reliable execution, the master thread manager keeps execution contexts of the migrated threads as a *checkpoint*, so it can consistently recover missing threads for an unintended device disconnection. After the checkpoint is created, non-migrated threads in the same application update memory pages in a copy-on-write manner to maintain original contents of shared pages. The checkpoint is updated only when the size of copied data exceeds a threshold (e.g., 20% of total memory size). As

[2] This conservative detection using the static threshold works well in DynaMix because the migration selector (§4.2.2) considers the trade-offs of all candidates and eventually decides the best migration target.

[3] DSM and I/O engine provide the information of data communications. The profiling overhead of each engine is typically insignificant because DSM engine measures the communication only in critical sections and I/O engine merely records the size of transferred data.

Algorithm 1: Migration Selector

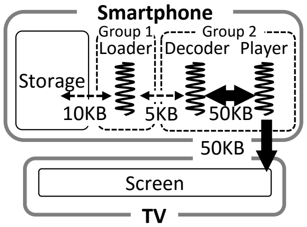
```
input    : the analyzed data communication result, $C$.
input    : a set of local threads, $threads_{local}$.
input    : a set of remote devices, $devices$.
output   : a tuple of a migration victim thread group to recommend,
           its destination device, and the network gain.
/* Construct thread groups                                  */
$tgroups = [[T]$ for $thread\ T$ in $threads_{local}]$
do
    /* Compare the amount of inter-thread comm.             */
    foreach $(tg1, tg2)$ where $tg1, tg2 \in tgroups$ do
        if $communication(tg1, tg2, C) > D_{thre}$ then
            $merge\_groups(tg1, tg2, tgroups)$
        end
    end
while $tgroups\ changed$;
/* Find a victim thread group and a destination device
   that yields the largest network gain                     */
$(victim\_tg, dest\_dev, max\_gain) = (null, null, 0)$
foreach $tg \in tgroups$ do
    /* Consider devices with enough idle CPU BW             */
    foreach $dev \in possible\_devices(tg, devices)$ do
        $net\_gain = estimate\_net\_gain(tg, dev, C)$
        if $net\_gain > max\_gain$ then
            $(victim\_tg, dest\_dev, max\_gain) = (tg, dev, net\_gain)$
        end
    end
end
return $(victim\_tg, dest\_dev, max\_gain)$
```

the threshold can affect memory pressure on a device, it is experimentally decided by considering an available memory size not to hurt other applications' performance.

4.2.2 Migration Selector

With the information delivered by a thread manager, the migration selector determines the best victim thread for migration and its destination device. The estimation relies on recent access patterns of an application with the assumption that similar behaviors appear in the near future. This assumption is reasonable in DynaMix's target applications which mainly access remote resources (e.g., repeatedly accessing a remote screen or camera) unless a user changes the resource configuration. The migration selector determines the best victim thread for migration and its destination device, and notifies the information to the master daemon as *migration recommendation*. Algorithm 1 describes how the migration selector finds the migration recommendation.

The migration selector first groups tightly coupled threads as a *thread group* which is a minimal migration unit. Such grouping simplifies the selection process and prevents unnecessary migration initiations. The algorithm sets threads as a thread group if their inter-thread communication amount is larger than a predefined threshold (D_{thre}). Next, it finds the best victim group and its destination device in a way to maximize the network overhead reduction, *network gain*. Note that the selected victim group is temporarily excluded in the next target selection during a specific time period to avoid frequent migration invocations on the same group. The time period is extended using exponential backoff.



(a) Thread grouping

Status	Value
$CPU_{\mathtt{thread}}$	30%
$idle_{tv}$	70%
$Perf_{tv,phone}$	0.9
$NAcq_{\mathtt{load,dec}}$	10
$lat_{tv,phone}$	0.01s
$BW_{tv,phone}$	100KB/s

(b) Device status

Thread Placement	T_{ct}	T_{dt}	T_{gain}
Group 1 on TV	-0.1 s	-0.15 s	-0.25 s
Group 2 on TV	-0.1 s	0.45 s	0.35 s

(c) Network gain estimation

Figure 4: Migration victim and destination selection

The destination device should have idle CPU bandwidth enough to accommodate the migrated threads. To consider the different CPU performance of devices, the algorithm uses a scaling factor, $Perf_{dest,source}$. For example, if $Perf_{dest,source}$ is 0.9, the destination device's CPU is slower than the source's CPU by 10%.

Calculating the network gain. The network gain T_{gain} quantifies how much the thread migration will improve the network performance in terms of the latency to transfer control messages (T_{ct}) and data (T_{dt}): $T_{gain} = T_{ct} + T_{dt}$.

A lock-acquire operation is the most critical source of the control messages, and each one incurs a three-hop latency (§4.2.1). The latency between thread i and j is the number of acquire operations ($NAcq_{i,j}$) times the three-hop latency between them ($lat_{D(i),D(j)}$), where $D(i)$ and $D(j)$ indicate the devices running thread i and j. Therefore, the total transfer latency is $\Sigma_{i \in tg} \Sigma_{j \in tcom} NAcq_{i,j} \times lat_{D(i),D(j)}$, where tg is the thread group and $tcom$ is a set of communicating threads. Then, the network gain of control message transfer, T_{ct}, is the latency difference due to the migration to the destination, dst:

$$T_{ct} = \sum_{i \in tg} \sum_{j \in tcom} NAcq_{i,j} \times (lat_{D(i),D(j)} - lat_{dst,D(j)})$$

The data transfer latency gain can also be calculated in a similar manner. If $D_{i,j}$ is the size of transferred data between thread i and j, and $BW_{Di,Dj}$ is the network bandwidth between them, the data transfer latency is $D_{i,j}/BW_{Di,Dj}$. Then, the network latency gain of data transfer, T_{dt}, is the latency difference due to the migration to the destination, dst:

$$T_{dt} = \sum_{i \in tg} \sum_{j \in tcom} (D_{i,j}/BW_{D(i),D(j)} - D_{i,j}/BW_{dst,D(j)})$$

Example Victim/Destination Selection Scenario. We illustrate example operations of the migration selector. Figure 4a shows the data communication status collected by the thread manager, and Figure 4b shows the status of devices collected by the device status monitor and the heartbeat communicator. Figure 4a also shows two thread groups, where the `Loader` and the `Decoder` thread are allocated in the same group because they heavily communicate each other. As the TV has enough idle CPU bandwidth ($70\% \times 0.9 = 63\%$) to accommodate either thread group (30% for Group 1, $30\% \times 2 = 60\%$ for Group 2), both groups can be migrated to the TV. Next, the migration selector compares the network gains of migrating either thread group (Figure 4c), and reports the best group and destination (i.e., Group 2 and TV) to the master daemon.

4.2.3 Migration Engine

After the migration selector decides the victim threads (i.e., thread group) and its destination device, the migration engine eventually performs a thread migration. DynaMix supports a low-overhead migration by adopting thread cloning and live migration. It minimizes the downtime, a suspended time period during migration, by transferring essential pages in a short time.

First, the source device sends only the memory layout (e.g., heap, stack) of the victim threads to the destination device which then creates their clone threads suspended during the migration. Next, for a short period (e.g., 2 secs), the migration engine transfers the most recently accessed pages (e.g., using the LRU-based page cache in Linux kernel) to the destination device, while the victim threads run on the original device. Using write permission faults (similar to §4.1.1), the migration engine detects and records the updated pages during the memory transfer. After finishing (i.e., timeout) the memory transfer, it sends the victim thread's execution context (e.g., process control blocks) with the updated pages in the meanwhile. This transparent live migration (similar to [20]) effectively hides the migration latency and minimizes the service downtime. Finally, the clone threads continue their execution on the destination device after the victim threads are suspended on the original device.

4.2.4 Heartbeat Communicator

For dynamic resource integration, DynaMix supports seamless operations whiles devices are plugged in and out. The heartbeat communicators periodically exchange heartbeat messages to check the device connectivity and share their resource status (e.g., CPU idleness, network latency, bandwidth). The resource status information is then delivered to the migration selector. Note that the inter-device network latency can be estimated from the round-trip latency of heartbeat messages.

The heartbeat communicator can detect which remote device joins or leaves a DynaMix device. For a newly joined device, its heartbeat communicator broadcasts heartbeat messages. On receiving the message, the master daemon enlists the new device in the DynaMix device. The heartbeat communicator also detects unstable devices by monitoring the connectivity (e.g., the number of packet drops). If a device becomes unstable, the heartbeat communicator notifies the master daemon to initiate migrating the threads in the device to more stable devices to avoid thread recovery that may cause the loss of the overall progress. For an unexpected disconnection. the master heartbeat communicator notifies the thread manager to recover from the latest checkpoint (§4.2.1).

4.3 Master Daemon

A DynaMix device has a single master daemon[4] that manages various system states (e.g., threads, locks, memory pages, files) to orchestrate DynaMix operations and components. The master daemon runs on the failure-free master device, and consists of three components: a thread directory, a page directory, and a file directory.

4.3.1 Thread Directory

The thread directory manages the global states of threads such as thread locations, and arbitrates the thread migration process. It collects resource contention signals from the device status monitors, and migration recommendations from the migration selectors. On receiving recommendations, the thread directory selects the best migration victim and its destination to achieve the highest network gain, and then manages the migration engines to perform the designed migrations.

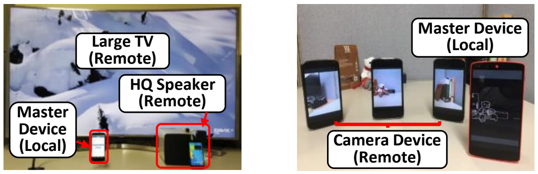
The thread directory also keeps the lock information (e.g., current owner, status). To acquire a lock, each device should consult the master device's thread directory. To reduce the lock acquisition overhead, the thread directory can speculatively grant the lock to frequent lock holding devices. When another device attempts to acquire the lock, the thread directory reclaims the speculatively given lock. Note that when a device is disconnected, the thread directory immediately reclaims all locks held by the device to avoid a deadlock.

4.3.2 Page Directory

The page directory manages the sharing state of memory pages to orchestrate memory synchronization operations. When a device sends a remote read request due to a page fault, the page directory consults a sharer table which keeps the sharer device lists of each page. It then relays the request to one of the sharer devices which will deliver the page to the requesting device.

On a lock release, the lock owner device reports the address list of updated pages to the page directory. In this way, the page directory identifies which pages should be sent to the next owner. When another device acquires the lock, the page directory manages its prior owner to forward the updated pages or their diffs if the new owner has old copies. Note that the transfers of shared pages

[4] Such a centralized approach enables easy management but might limit scalability. We believe that composing multiple DynaMix devices rather than a single large one is much preferable in our scenarios.



(a) Home Theater (b) Smart Surveillance

Figure 5: Two example DynaMix applications

mostly occur when a device newly acquires a lock due to LRC memory model.

4.3.3 File Directory

The file directory manages the file metadata and the physical locations of shared files for the globally consistent file view. Whenever a new device joins the DynaMix device or a device updates the metadata, the master daemon updates its file directory and then notifies the updated information to other devices. Note that except the device which owns a file, each device holds the file's read-only copy in memory.

5 Evaluation

5.1 Experimental Setup

We implemented our example DynaMix prototype which can be easily installed on top of existing Android and Tizen devices. For our evaluation, we installed DynaMix on Google Nexus smartphones (i.e., Nexus 4 and 5) and an in-house Samsung Smart TV. The smartphones run Android 5.1.1 (CyanogenMod 12.1) with Linux kernel version 3.4 patch, and the Samsung Smart TV runs Tizen 2.3 with Linux kernel version 3.0 patch. All devices are connected to the same Wi-Fi network (IEEE 802.11ac) with the maximum bandwidth of 100Mbps.

To evaluate the DynaMix prototype, we introduce three example multi-device use cases (i.e., home theater, smart surveillance, and photo classification) designed to utilize both computation and I/O resources simultaneously. We believe users can easily make other interesting services using our framework.

Home Theater. The home theater is a typical multi-threaded movie player application which loads and decodes a movie file from a storage, shows the video on a screen, and plays the audio through a speaker. DynaMix allows users to configure resources (e.g., a large TV, an HQ speaker) used to run the home theater. Figure 5a shows the example home theater setup with three devices. We used FFmpeg [4] to decode video and audio frames. Here, the home theater plays a movie with both video and audio frames synchronized.

Smart Surveillance. The smart surveillance is another possible service that performs image processing (e.g., edge detection) with preview images from a remote camera. Figure 5b shows the example smart surveillance setup using four devices. A processing thread performs edge detection on preview images from a selected camera device, and a UI thread displays the processed image on the screen. We used the Canny edge detection algorithm [16] to detect moving objects in the service.

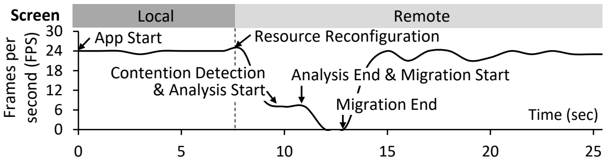


Figure 6: Perf. timeline of the home theater application

Photo Classification. Lastly, integrating storage resources enables a shared storage system across devices where users can observe scattered remote files (e.g., photos, videos) in the same hierarchy and easily access them at any device. To evaluate the storage system, we perform object classifications for photos scattered in the connected devices. For the purpose, we used an object classifier with the pre-trained CNN model (SSD_MobileNet [31]) using a TensorFlow [9] library. Each thread classifies its assigned photos with the classifier and reports the results to the collector thread.

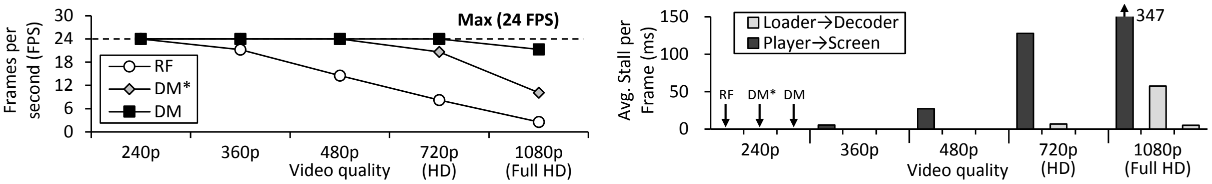
5.2 Operation Models

This evaluation revisits the basic operation models of DynaMix in §3.2. We use the multi-threaded home theater with loader, decoder, and player threads. It runs on the DynaMix device (Figure 5a) configured with a Samsung Smart TV as the remote screen, an HQ speaker attached to Nexus 4 as the remote audio, and a Nexus 5 smartphone as the master device. We play an HD (720p) movie and measure its frames per second (FPS).

Figure 6 shows its performance timeline. When the user initially plays a movie on the master device, the home theater displays video frames on the local smartphone screen with the target performance of 24 FPS. However, after the user suddenly switches the screen device to the remote TV (8 sec), it suffers from significant FPS drops due to the huge network traffic caused by forwarding HD video frames to the TV. Therefore, DynaMix immediately detects a network contention (10 sec), decides the best task redistribution plan (11 sec), and migrates the video decoder and player threads to the TV (13 sec). Although the performance temporarily degrades due to the increased network consumption caused by the migration, DynaMix quickly restores the target performance (14 sec) with a negligible service downtime. This experiment verifies that DynaMix enhances the service quality with resource-aware task redistribution even in the sudden resource reconfiguration.

5.3 Service Quality

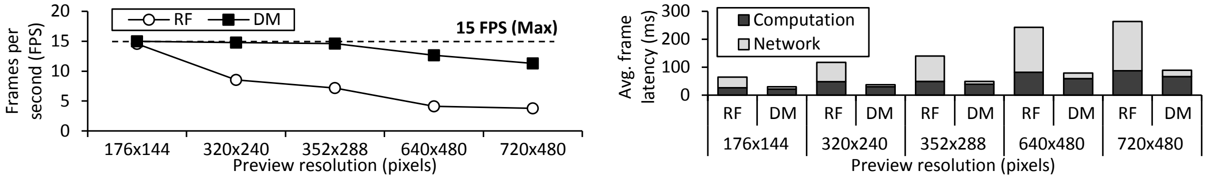
We now evaluate three use cases (i.e., home theater, smart surveillance, and photo classification) to verify that



(a) Throughput (FPS)

(b) Frame-average network stall time

Figure 7: The home theater performance for Request Forwarding (RF) and DynaMix (DM) (*: no prefetching)



(a) Throughput (FPS)

(b) Frame-average detection latency breakdown

Figure 8: The smart surveillance performance for Request Forwarding (RF) and DynaMix (DM)

DynaMix significantly enhances the service quality. To show the benefit of resource-aware task redistribution, we compare DynaMix with Request Forwarding (RF) as a representative baseline because its resource sharing mechanism conceptually includes the state-of-the-art work (e.g., Rio [11]), which accesses a remote resource and receives the result via a wireless network. We also evaluate DynaMix without memory prefetching.

Home Theater. We configure a DynaMix device as explained in §5.2, and measure the throughput (i.e., frames per second) of the home theater on the DynaMix device. RF forwards decoded frames to the remote device because all threads run on the master device.

Figure 7a compares the throughput (FPS) of the home theater for RF and DynaMix. While RF suffers from increasing throughput degradations with the target video quality improved, DynaMix successfully achieves the target throughput up to the decent quality (480p) even without memory prefetching. Enabling the prefetching further enhances the throughput, which makes DynaMix achieve the throughput close to the maximum for the full HD quality (1080p). DynaMix achieves 8.3x higher throughput than RF, while paying only 11% performance drop from the maximum throughput for 1080p.

Figure 7b compares the network stall time of three design points to process a video frame for the various video qualities. The network stall time means how much network traffic affects the per-frame latency, and helps to clearly investigate why RF suffers from the low throughput. First, RF incurs severe network traffic to transfer a decoded frame between the player thread and the remote TV screen even for a relatively inferior quality (360p). Moreover, RF suffers from a huge amount of network stall as the video quality increases. On the other hand, as DynaMix can migrate the video decoder and player threads to the TV, the loader thread on the master device can timely transfer small-sized encoded frames to the decoder threads on the TV. As a result, DynaMix effectively hides the network stall up to 480p, and applying the memory prefetching further amortizes the network overheads (i.e., near-zero network stall for 1080p).

Smart Surveillance. We configure a DynaMix device to use a Nexus 5 device as a master device with a screen, and three Nexus 4 devices as remote cameras, as shown in Figure 5b. As this application allows a user to select a target remote camera, we randomly choose one camera as the current input feeder. RF receives preview images from the remote camera because a processing thread runs on the master device. We now assume that DynaMix is equipped with memory prefetching by default.

Figure 8a compares the throughput (FPS) of the smart surveillance for RF and DynaMix. While RF suffers from significant throughput degradation with the preview resolution increased, DynaMix retains moderate performance drop as only 24.7% compared with the target throughput for the highest preview resolution (720x480). Figure 8b shows the breakdown of the average latency in detecting edges of a preview image. After analyzing the tradeoff, DynaMix migrates the edge-detector threads to the camera devices to avoid the network contention. As a result, it achieves far less network latency than forwarding raw preview images from the camera to the master device. Note that the computation still occupies a significant portion of the total latency due to the lack of sufficient computation resources. This result suggests deploying faster CPUs on remote cameras so that DynaMix can completely remove the computation overhead.

Photo Classification. We configure a DynaMix device to use up to four Nexus 5 smartphones to construct the shared storage system. Each device has 100MB of photos with different sizes ranging from 4KB to 10MB. A user can choose the number of classifier threads and

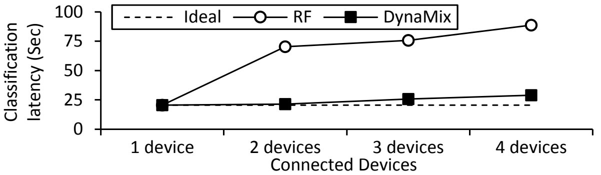


Figure 9: The total latency of the photo classification for all photo files on the shared storage system

	Remote File Size (B)				
	<10K	<100K	<1M	<5M	≤10M
RF (%)	87.8	64.4	33.3	13.9	0
Mig. (%)	12.2	35.6	66.7	86.1	100

Table 1: DynaMix's request forwarding (RF) vs. migration (Mig.) ratio on the photo classification

would launch threads in proportion to the total size of photos. In our four-device configuration, we assume that four threads classify total 400MB of photos.

We then measure the latency to perform the classification for all photos, and compare the performance of RF and DynaMix. We also mark the ideal performance to identify the bottleneck. We assume that the ideal one classifies all files on the remote devices without any network overheads. Note that RF forwards remote files to the threads running on the master device.

Figure 9 compares the performance of the total classification latency. As the number of connected devices increases, RF suffers from high latency due to the increasing network overheads incurred by forwarding remote files to the classifiers. On the other hand, DynaMix is barely affected by the network overheads and thus achieves the latency close to the ideal one, even for the four-device configuration. It is because DynaMix dynamically redistributes the threads across devices to minimizes the network overheads.

Furthermore, DynaMix can dynamically use either of request forwarding and the adaptive task migration, based on their tradeoffs. Note that the migrated threads should use request forwarding during a certain time period to prevent frequent migrations (§4.2.2). To emphasize the point, Table 1 shows the percentage of the two cases in the four-device configuration. DynaMix is likely to use the request forwarding more for small files (i.e., <100KB) to avoid the migration overhead, whereas it is likely to migrate threads for large-sized files (i.e., >1MB) to avoid the transfer overhead.

5.4 Network Sensitivity

To identify the performance impact for a given network bandwidth, we measure the performance of the home theater by playing an HD movie on RF and DynaMix. For this experiment, we vary the available bandwidth with Linux `tc` utility, and measure the average FPS as the performance metric.

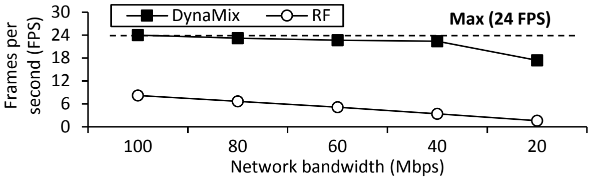


Figure 10: The home theater performance on various network bandwidth to play an HD video

	Average Power (mW)		
	Master Device	Screen Device	Total
RF	4985.90	5151.37	10137.27
DynaMix	2956.55	6480.51	9427.06

Table 2: The power consumption of the home theater for Request Forwarding (RF) and DynaMix

Figure 10 shows that RF severely suffers from its performance drops even with the maximum network bandwidth available (100Mbps) and further as the network bandwidth decreases. On the other hand, DynaMix maintains the target 24 FPS with only 40Mbps of bandwidth available. This result indicates that DynaMix effectively minimizes the network overhead by adaptively redistributing tasks among devices.

5.5 Power Consumption

In this experiment, we measure the power consumption of DynaMix while playing an HD movie clip, and compare it against RF. To measure the impact of inter-device traffic reduction, we use two Nexus 5 smartphones as a master device and a screen device. We use Monsoon power monitor [5] to measure the power consumption.

Table 2 measures the power consumption of the devices. First, the master device consumes much less power with DynaMix than RF by migrating a rendering task to the remote device and thus reducing the network traffic. On the other hand, the screen device consumes little more power with DynaMix than RF by running a relocated rendering task. As a result, DynaMix reduces the total power by 7% mainly due to the reduced network overhead. More importantly, as DynaMix's service quality (or performance) is 3-4 times higher than RF (Figure 10) and their power consumptions are similar (Table 2), DynaMix's overall energy efficiency can be considered 3-4 times higher for the target service quality. For further energy reduction, DynaMix may redistribute tasks in a way to maximize the energy efficiency.

6 Discussion

Heterogeneous ISA/OS. One interesting issue related to our work is to extend the coverage of architecture and operating system used by DynaMix devices. However, it is a well-known challenge to seamlessly share resources in heterogeneous devices using different ISAs and OSes. Therefore, existing work often assume either homogeneous ISA/OS [11,43] or expensive VM supports to emulate the homogeneous platform [19,21,28].

In this work, DynaMix also assumes homogeneous ISA/OS environments for the most popular mobile platform (i.e., Andriod/ARM). However, we showed that its OS coverage can be easily extended to Tizen which shares the Linux kernel base. To improve the platform coverage further, we believe that the following directions seem to be promising. For non-Linux based OSes (e.g., iOS), DynaMix may implement a compatibility layer between kernel and application by taking approaches similar to existing OS-compatibility schemes [8,12]. To support cross-ISA (e.g., ARM to x86) migrations, DynaMix may implement a native offloading using the compiler-assisted method [39] or dynamic binary translation [51].

Developing DynaMix Applications. To fully utilize DynaMix's resource-aware task redistribution, programmers are recommended to compose applications with multiple threads, specialized in certain computing jobs or I/O resource accesses. We believe this guideline is not burdensome to programmers, as many recent programming conventions also recommend similar guidelines for optimal performance [1, 7]. To further accommodate easy application development, DynaMix may adopt existing automatic code parallelization techniques [34, 40, 49] to maximize the effectiveness of resource-aware task redistribution without additional programming effort.

Security Concerns. Another assumption of this work is that a user shares resources in only user-owned trusted devices, as existing schemes such as task offloading [19, 21, 28] and remote IO forwarding [11]. In fact, resource sharing with untrusted devices is not common scenarios that DynaMix considers. Therefore, the security issues related to untrusted devices are beyond the scope of our work. However, we believe that DynaMix can resolve such security issues by adopting existing secure task-offloading schemes [26,42,45,47], without a noticeable increase in complexity.

7 Related Work

Cross-device Resource Sharing. Single system image (SSI) [17, 18, 25] is traditional work to integrate resources by creating one single system with a cluster of machines connected to a fast and stable wired network. However, its complex operations and huge synchronization overheads are not suitable to the mobile environment with limited communication capabilities. Thus, similar studies in mobile computing have focused on how to selectively integrate remote resources. For example, offloading schemes [19,21,28] offload compute-intensive tasks to powerful servers, even for heterogeneous ISAs [39, 51]. Solutions to utilize other resources such as GPU [22], screen [14], storage [23, 44, 46], generic I/O resources [11] and platform-level services [43] have also been proposed. While they only support specific types of resources, DynaMix integrates a wide spectrum of computation and I/O resources. On the other hand, some studies [50, 53] have optimized spectrum utilization sharing in cellular networks. Such techniques are orthogonal to our work but we can adopt them to more efficiently communicate between devices.

Multi-device Programming Platform. To facilitate easy application development in the multi-device environment, [54] allows programmers to develop unit objects and automatically deploys them across devices. [27] also provides a DSM platform and APIs for multi-device applications. Such platforms, however, still force programmers to explicitly partition applications with special APIs. [24] provides a control interface to access various home appliances with unified APIs, but it does not distribute tasks for efficient resource utilization. DynaMix, on the other hand, enables task redistributions of single-device applications for multi-device services, without explicit application partitioning.

Thread Migration. The thread migration is a widely supported feature in distributed computing platforms [13,41,52,55]. Especially, to reduce a service downtime during migration, various VM platforms [15,29,36] have implemented the pre-copy [20] or the post-copy [30,35] live migrations, depending on the timing to send execution contexts. DynaMix also applies such live migration schemes to our environment. Researchers have proposed an online thread distribution algorithm [48] to minimize inter-thread network overheads. However, DynaMix resolves CPU contention as well as network contention, optimized to the mobile environment.

8 Conclusion

In the era of the Internet of Things, a user can access an increasing number of heterogeneous devices. We proposed DynaMix, a novel framework to enable efficient cross-device resource sharing by integrating diverse resources and dynamically redistribute tasks. Our example implementation on the top of Android and Tizen devices showed that DynaMix can efficiently support multi-device services using single-device applications.

Acknowledgment

This work was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2015M3C4A7065647, NRF-2017R1A2B3011038), and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. R0190-15-2012). We also appreciate the support from Automation and Systems Research Institute (ASRI) and Inter-University Semiconductor Research Center (ISRC) at Seoul National University.

References

[1] Android developer training - sending operations to multiple threads. `https://developer.android.com/training/multiple-threads/index.html`.

[2] Android IP Webcam. `https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en`.

[3] Android Wi-Fi Speaker. `https://play.google.com/store/apps/details?id=pixelface.android.audio&hl=en`.

[4] FFmpeg. `https://ffmpeg.org/`.

[5] Monsoon Solutions Inc. Monsoon Power Monitor. `http://www.msoon.com/`.

[6] Nest cam. `https://www.nest.com/camera/meet-nest-cam/`.

[7] Tizen development guide - using threads. `https://developer.tizen.org/development/guides/native-application/user-interface/efl/core-loop-and-os-interfacing/using-threads?langredirect=1`.

[8] Wine. `https://www.winehq.org/`.

[9] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[10] Amiri Sani, A., Boos, K., Qin, S., and Zhong, L. I/o paravirtualization at the device file boundary. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014).

[11] Amiri Sani, A., Boos, K., Yun, M. H., and Zhong, L. Rio: A system solution for sharing I/O between mobile systems. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (2014).

[12] Andrus, J., Hof, A. V., AlDuaij, N., Dall, C., Viennot, N., and Nieh, J. Cider: Native execution of iOS apps on Android. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014).

[13] Aridor, Y., Factor, M., and Teperman, A. cJVM: A single system image of a JVM on a cluster. In *Proceedings of the International Conference on Parallel Processing* (1999).

[14] Baratto, R. A., Kim, L. N., and Nieh, J. THINC: A virtual display architecture for thin-client computing. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (2005).

[15] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, a. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003).

[16] Canny, J. A computational approach to edge detection. *IEEE Transactions On Pattern Analysis and Machine intelligence 8*, 6 (1986).

[17] Chapin, J., Rosenblum, M., Devine, S., Lahiri, T., Teodosiu, D., and Gupta, A. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995).

[18] Cheriton, D. The v distributed system. *Commun. ACM 31*, 3 (1988).

[19] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. CloneCloud: Elastic execution between mobile device and cloud. In *Proceedings of the 6th Conference on Computer Systems* (2011).

[20] Clark, C., Fraser, K., Hand, S., and Hansen, J. G. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation* (2005).

[21] Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. MAUI: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (2010).

[22] Cuervo, E., Wolman, A., Cox, L. P., Lebeck, K., Razeen, A., Saroiu, S., and Musuvathi, M. Kahawai: High-quality mobile gaming using GPU offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (2015).

[23] Dahlin, M., Gao, L., Nayate, A., Venkataramana, A., Yalagandula, P., and Zheng, J. PRACTI replication. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation* (2006).

[24] Dixon, C., Mahajan, R., Agarwal, S., Brush, A., Lee, B., Saroiu, S., and Bahl, P. An operating system for the home. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012).

[25] Feeley, M. J., Morgan, W. E., Pighin, E. P., Karlin, A. R., Levy, H. M., and Thekkath, C. A. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995).

[26] Feldman, A. J., Zeller, W. P., Freedman, M. J., and Felten, E. W. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010).

[27] Gao, J., Sivaraman, A., Agarwal, N., Li, H., and Peh, L.-S. DIPLOMA: Consistent and coherent shared memory over mobile phones. In *Proceedings of the 30th International Conference on Computer Design* (2012).

[28] Gordon, M. S., Jamshidi, D. A., Mahlke, S., Mao, Z. M., and Chen, X. COMET: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012).

[29] Gulati, A., Shanmuganathan, G., Holler, A., Waldspurger, C., Ji, M., and Zhu, X. VMware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal* (2012).

[30] Hines, M. R., and Gopalan, K. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2009).

[31] Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., Fischer, I., Wojna, Z., Song, Y., Guadarrama, S., and Murphy, K. Speed/accuracy trade-offs for modern convolutional object detectors. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017).

[32] Keleher, P., Cox, A. L., and Zwaenepoel, W. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (1992).

[33] Kim, B., Heo, S., Lee, G., Park, S., Kim, H., and Kim, J. Heterogeneous Distributed Shared Memory for Lightweight Internet of Things Devices. *IEEE Micro 36*, 6 (2016).

[34] Kim, H., Johnson, N. P., Lee, J. W., Mahlke, S. A., and August, D. I. Automatic speculative doall for clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (2012).

[35] Kim, J., Chae, D., Kim, J., and Kim, J. Guide-copy: Fast and silent migration of virtual machine for datacenters. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013).

[36] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. KVM: The Linux virtual machine monitor. In *Proceedings of The Ottawa Linux Symposium* (2007).

[37] Lee, G., Heo, S., Kim, B., Kim, J., and Kim, H. Integrated IoT Programming with Selective Abstraction. In *Proc. 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)* (2017).

[38] Lee, G., Heo, S., Kim, B., Kim, J., and Kim, H. Rapid prototyping of IoT applications with Esperanto compiler. In *Proc. 28th International Symposium on Rapid System Prototyping (RSP)* (2017).

[39] Lee, G., Park, H., Heo, S., Chang, K.-A., Lee, H., and Kim, H. Architecture-aware automatic computation offload for native applications. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015).

[40] Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., and Torrellas, J. Posh: A tls compiler that exploits program structure. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2006).

[41] Ma, M. J., Wang, C.-L., and Lau, F. C. Delta Execution: A preemptive Java thread migration mechanism. *Cluster Computing 3*, 2 (2000), 83–94.

[42] Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., and Walfish, M. Depot: Cloud storage with minimal trust. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010).

[43] Oh, S., Yoo, H., Jeong, D. R., Bui, D. H., and Shin, I. Mobile plus: Multi-device mobile platform for cross-device functionality sharing. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services* (2017).

[44] Perkins, D., Agrawal, N., Aranya, A., Yu, C., Go, Y., Madhyastha, H. V., and Ungureanu, C. Simba: Tunable end-to-end data consistency for mobile apps. In *Proceedings of the 10th European Conference on Computer Systems* (2015).

[45] Portokalidis, G., Homburg, P., Anagnostakis, K., and Bos, H. Paranoid Android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference* (2010).

[46] Ramasubramanian, V., Rodeheffer, T. L., Terry, D. B., Walraed-Sullivan, M., Wobber, T., Marshall, C. C., and Vahdat, A. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (2009).

[47] Shraer, A., Cachin, C., Cidon, A., Keidar, I., Michalevsky, Y., and Shaket, D. Venus: Verification for untrusted cloud storage. In *Proceedings of the ACM Workshop on Cloud Computing Security Workshop* (2010).

[48] Thitikamol, K., and Kelenher, P. Thread migration and communication minimization in DSM systems. *Proceedings of The IEEE 87*, 3 (1999), 487–497.

[49] Vachharajani, N., Rangan, R., Raman, E., Bridges, M. J., Ottoni, G., and August, D. I. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques* (2007).

[50] Wang, J., Zhu, D., Zhao, C., Li, J. C., and Lei, M. Resource sharing of underlaying device-to-device and uplink cellular communications. *IEEE Communications Letters 17*, 6 (2013), 1148–1151.

[51] Wang, W., Yew, P.-C., Zhai, A., McCamant, S., Wu, Y., and Bobba, J. Enabling cross-isa offloading for cots binaries. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services* (2017).

[52] Wood, T., Shenoy, P., Venkataramani, A., and Yousif, M. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation* (2007).

[53] Yu, C.-H., Doppler, K., Ribeiro, C. B., and Tirkkonen, O. Resource sharing optimization for device-to-device communication underlaying cellular networks. *IEEE Transactions on Wireless communications 10*, 8 (2011), 2752–2763.

[54] Zhang, I., Szekeres, A., Van Aken, D., Ackerman, I., Gribble, S. D., Krishnamurthy, A., and Levy, H. M. Customizable and extensible deployment for mobile/cloud applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014).

[55] Zhu, W., Wang, C.-L., and Lau, F. C. M. JESSICA2: A distributed Java virtual machine with transparent thread migration support. In *Proceedings of the IEEE International Conference on Cluster Computing* (2002).

The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS

Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel
EPFL

Redha Gouicem, Julia Lawall, Gilles Muller, Julien Sopena
Sorbonne University, Inria, LIP6

Abstract

This paper analyzes the impact on application performance of the design and implementation choices made in two widely used open-source schedulers: ULE, the default FreeBSD scheduler, and CFS, the default Linux scheduler. We compare ULE and CFS in otherwise identical circumstances. We have ported ULE to Linux, and use it to schedule all threads that are normally scheduled by CFS. We compare the performance of a large suite of applications on the modified kernel running ULE and on the standard Linux kernel running CFS. The observed performance differences are solely the result of scheduling decisions, and do not reflect differences in other subsystems between FreeBSD and Linux.

There is no overall winner. On many workloads the two schedulers perform similarly, but for some workloads there are significant and even surprising differences. ULE may cause starvation, even when executing a single application with identical threads, but this starvation may actually lead to better application performance for some workloads. The more complex load balancing mechanism of CFS reacts more quickly to workload changes, but ULE achieves better load balance in the long run.

1 Introduction

Operating system kernel schedulers are responsible for maintaining high utilization of hardware resources (CPU cores, memory, I/O devices) while providing fast response time to latency-sensitive applications. They have to react to workload changes, and handle large numbers of cores and threads with minimal overhead [12]. This paper provides a comparison between the default schedulers of two of the most widely deployed open-source operating systems: the Completely Fair Scheduler (CFS) used in Linux, and the ULE scheduler used in FreeBSD. Our goal is *not* to declare an overall winner. In fact, we find that for some workloads ULE is better and for others CFS is better. Instead, our goal is to illustrate how differences in the design and the implementation of the two schedulers are reflected in application performance under different workloads.

ULE and CFS are both designed to schedule large numbers of threads on large multicore machines. Scalability considerations have led both schedulers to adopt per-core runqueues. On a context switch, a core accesses only its local runqueue to find the next thread to run. Periodically and at select times, e.g., when a thread wakes up, both ULE and CFS perform load balancing, i.e., they try to balance the amount of work waiting in the runqueues of different cores.

ULE and CFS, however, differ greatly in their design and implementation choices. FreeBSD ULE is a simple scheduler (2,950 lines of code in FreeBSD 11.1), while Linux CFS is much more complex (17,900 lines of code in the latest LTS Linux kernel, Linux 4.9). FreeBSD runqueues are FIFO. For load balancing, FreeBSD strives to even out the number of threads per core. In Linux, a core decides which thread to run next based on prior execution time, priority, and perceived cache behavior of the threads in its runqueue. Instead of evening out the number of threads between cores, Linux strives to even out the average amount of pending work.

The major challenge in comparing ULE and CFS is that application performance depends not only on the scheduler, but also on other OS subsystems, such as networking, file systems and memory management, which also differ between FreeBSD and Linux. To isolate the effect of differences between CFS and ULE, we have ported ULE to Linux, and we use it as the default scheduler to run all threads on the machine (including kernel threads that are normally scheduled by CFS). Then, we compare application performance between this modified Linux with ULE and the default Linux kernel with CFS.

We first examine the impact of the per-core scheduling decisions made by ULE and CFS, by running applications and combinations of applications on a single core, We then run the applications on all cores of the machine, and study the impact of the load balancer. We use 37 applications ranging from scientific HPC applications to databases. While offering similar performance in many circumstances, CFS and ULE occasionally behave very differently, even on simple workloads consisting of one application running one thread per core.

This paper makes the following contributions:

- We provide a complete port of FreeBSD's ULE scheduler in Linux and release it as open source [21]. Our implementation contains all the features and heuristics used in the FreeBSD 11.1 version.
- We compare the application performance under the ULE and CFS schedulers in an otherwise identical environment.
- Unlike CFS, ULE may starve threads that it deems non-interactive for an unbounded amount of time. Surprisingly, starvation may also occur when the system executes only a single application consisting of identical threads. Even more surprising, this behavior actually proves beneficial for some workloads (e.g., a database workload).
- CFS converges faster towards a balanced load, but ULE achieves a better load balance in the long run.
- The heuristics used by CFS to avoid migrating threads can hurt performance in HPC workloads that only use one thread per core, because CFS sometimes places two threads on the same core, while ULE always places one thread on each core.

The outline of the rest of this paper is as follows. Section 2 presents the CFS and ULE schedulers. Section 3 describes our port of ULE to Linux and the main differences between the native ULE implementation and our port. Sections 4 presents the machines and the workloads used in our experiments. Section 5 analyzes the impact of per-core scheduling in CFS and ULE. Section 6 analyzes the load balancer of CFS and ULE. Section 7 presents related work and Section 8 concludes.

2 Overview of CFS and ULE

2.1 Linux CFS

Per-core scheduling: Linux's CFS implements a weighted fair queueing algorithm: it evenly divides CPU cycles between threads weighted by their priority (represented by their niceness, high niceness meaning low priority and vice versa) [18]. To that end, CFS orders threads by a multi-factor value called *vruntime*, representing the amount of CPU time a thread has already used divided by the thread's priority. Threads with the same priority and same vruntime have executed the same amount of time, meaning that core resources have been shared fairly between them. To ensure that the vruntime of all threads progresses fairly, when the current running thread is preempted, CFS schedules the thread with the lowest vruntime to run next.

Since Linux 2.6.38 the notion of fairness in CFS has evolved from fairness between threads to fairness between applications. Before Linux 2.6.38 every thread was considered as an independent entity and got the same share of resources as other threads in the system. This meant that an application that used many threads got a larger share of resources than single-threaded applications. In more recent kernel versions, threads of the same application are grouped into a structure called a *cgroup*. A cgroup has a vruntime that corresponds to the sum of the vruntimes of all of its threads. CFS then applies its algorithm on cgroups, ensuring fairness between groups of threads. When a cgroup is chosen to be scheduled, its thread with the lowest vruntime is executed, ensuring fairness within a cgroup. Cgroups can also be nested. For instance, systemd automatically configures cgroups to ensure fairness between different users, and then fairness between the applications of a given user.

CFS avoids thread starvation by scheduling all threads within a given time period. For a core executing fewer than 8 threads the default time period is 48ms. When a core executes more than 8 threads, the time period grows with the number of threads and is equal to $6 * number\ of\ threads$ ms; the 6ms value was chosen to avoid preempting threads too frequently. Threads with a higher priority get a higher share of the time period. Since CFS schedules the thread with the lowest vruntime, CFS needs to prevent a thread from having a vruntime much lower than the vruntimes of the other threads waiting to be scheduled. If that were to happen, the thread with the low vruntime could run for a long time, starving the other threads. In practice, CFS ensures that the vruntime difference between any two threads is less than the preemption period (6ms). It does so at two key points: (i) when a thread is created, the thread starts with a vruntime equal to the maximum vruntime of the threads waiting in the runqueue, and (ii) when a thread wakes up after sleeping, its vruntime is updated to be at least equal to the minimum vruntime of the threads waiting to be scheduled. Using the minimum vruntime also ensures that threads that sleep a lot are scheduled first, a desirable strategy on desktop systems, because it minimizes the latency of interactive applications. Most interactive applications sleep most of the time, waiting for user input, and are immediately scheduled as soon as the user interacts with them.

CFS also uses heuristics to improve cache usage. For instance, when a thread wakes up, it checks the difference between its vruntime and the vruntime of the currently running thread. If the difference is not significant (less than 1ms), the current running thread is not preempted – CFS sacrifices latency to avoid frequent thread preemption, which may negatively impact caches.

Load balancing: In a multicore setting, Linux's CFS evens out the amount of *work* on all cores of the machine. This is different from evening out the number of threads. For instance, if a user runs 1 CPU-intensive thread and 10 threads that mostly sleep, CFS might schedule the 10 mostly sleeping threads on a single core.

To balance the amount of work, CFS uses a *load* metric for threads and cores. The load of a thread corresponds to the average CPU utilization of a thread: a thread that never sleeps has a higher load than one that sleeps a lot. Like the vruntime, the load of a thread is weighted by the thread's priority. The load of a core is the sum of the loads of the threads that are runnable on that core. CFS tries to even out the load of cores.

CFS takes into account the loads of cores when creating or waking up threads. The scheduler first decides which cores are suitable to host the thread. This decision involves many heuristics, such as the frequency at which the thread that initiated the wakeup wakes up threads. For instance, if CFS detects a 1-to-many producer-consumer pattern, then it spreads out the consumer threads as much as possible on the machine, and most cores of the machine are considered suitable to host woken up threads. In a 1-to-1 communication pattern, CFS restricts the list of suitable cores to cores sharing a cache with the thread that initiated the wakeup. Then, among all suitable cores, CFS chooses the core with the lowest load on which to wake up or create the thread.

Load balancing also happens periodically. Every 4ms every core tries to steal work from other cores. This load balancing takes into account the topology of the machine: cores try to steal work more frequently from cores that are "close" to them (e.g., sharing a cache) than from cores that are "remote" (e.g., on a remote NUMA node). When a core decides to steal work from another core, it tries to even out the load between the two cores by stealing as many as 32 threads from the other core. Cores also immediately call the periodic load balancer when they become idle.

On large NUMA machines, CFS does not compare the load of all cores against each other, but instead balances the load in a hierarchical way. For instance, on a machine with 2 NUMA nodes, CFS balances the load of cores inside the NUMA nodes, and then compares the load of the NUMA nodes (defined as the average load of their cores) to decide whether or not to balance the load between nodes. If the load difference between the nodes is small (less than 25% in practice), then no load balancing is performed. The greater the distance between two cores (or groups of cores), the higher the imbalance has to be for CFS to balance the load.

2.2 FreeBSD ULE

Per-core scheduling: ULE uses two runqueues to schedule threads: one runqueue contains *interactive* threads, and the other contains *batch* threads. A third runqueue called *idle* is used when a core is idle. This runqueue only contains the idle task.

The goal of having two runqueues is to give priority to interactive threads. Batch processes usually execute without user interaction, and thus scheduling latency is less important. ULE keeps track of the *interactivity* of a thread using an *interactivity penalty* metric between 0 and 100. This metric is defined as a function of the time r a thread has spent running and the time s a thread has spent voluntarily sleeping (not including the time spent waiting for the CPU), and is computed as follows:

$$scaling\,factor = m = 50$$

$$penalty(r,s) = \begin{cases} \frac{m}{\frac{s}{r}} & s > r \\ \frac{m}{\frac{r}{s}} + m & otherwise \end{cases}$$

A penalty in the lower half of the range (≤ 50) means that a thread has spent more time voluntarily sleeping than running, while a penalty above means the opposite.

The amount of history kept for the sleep and running times is (by default) limited to the last 5 seconds of the thread's lifetime. On the one hand, having a large amount of history would lengthen the time required to detect batch threads. On the other hand, too little history would induce noise in the classification [15].

To classify threads, ULE first computes a *score* defined as *interactivity penalty* + *niceness*. A thread is considered interactive if its score is under a certain threshold, 30 by default as in FreeBSD11.1. With a niceness value of 0, this corresponds roughly to spending more than 60% of the time sleeping. Otherwise, it is classified as batch. A negative *nice* value (high priority) makes it easier for a thread to be considered interactive.

When a thread is created, it inherits the runtime and sleeptime (and thus the interactivity) of its parent. When a thread dies, its runtime in the last 5 seconds is returned to its parent. This penalizes parents that spawn batch children when being interactive.

Inside the interactive and batch runqueues, threads are further sorted by *priority*. The priority of interactive threads is a linear interpolation of their *score* (i.e., a thread with a penalty of 0 has the highest interactive priority, while a thread with a penalty of 30 has the lowest interactive priority). Inside the interactive runqueue, there is one FIFO per priority. To add a thread to a runqueue, the scheduler inserts the thread at the end of the FIFO indexed by the thread's priority. Picking a thread to run from this runqueue is simply done by taking the first thread in the highest-priority non-empty FIFO.

The priority of batch threads depends on their runtime: the more a thread runs, the lower its priority. The niceness of the thread is added to get a linear effect on the priority. Inside the batch runqueue, there is also one FIFO per priority. Insertion and removal work as in the interactive case, with a slight difference. To avoid starvation between batch threads, ULE tries to be fair among batch threads by minimizing the difference of runtime between threads, similarly to what CFS does with the vruntime.

When picking the next thread to run, ULE first searches in the interactive runqueue. If an interactive thread is ready to be scheduled, it returns that thread. If the interactive runqueue is empty, ULE searches in the batch runqueue instead. If both runqueues are empty, that means that the core is idle, and no thread is scheduled.

The order in which ULE searches the runqeues effectively gives interactive threads absolute priority over batch threads. Batch threads may potentially starve if the machine executes too many interactive threads. However, it is thought that, as interactive threads by definition sleep more than they execute, starvation does not occur.

A thread runs for a limited period of time defined as a *timeslice*. Contrary to CFS, the rate at which a thread's timeslice expires is the same regardless of its priority. However, the value of a timeslice changes with the number of threads currently running on the core. When a core executes 1 thread, the timeslice is 10 ticks (78ms). When multiple threads are running, this value is divided by the number of threads while being constrained to a lower bound of 1 tick (1/127th of a second). In ULE, full preemption is disabled, meaning that only kernel threads can preempt others.

Load balancing: ULE only aims to even out the number of threads per core. In ULE, the load of a core is simply defined as the number of threads currently runnable on this core. Unlike CFS, ULE does not group threads into cgroups, but rather considers each thread as an independent entity.

When choosing a core for a newly created or awoken thread, ULE uses an affinity heuristic. If the thread is considered cache affine on the last core it ran on, then it is placed on this core. Otherwise, ULE finds the highest level in the topology that is considered affine, or the entire machine if none is available. From there, ULE first tries to find a core on which the minimum priority is higher than that of this thread. If that fails, ULE tries again, but now on all cores of the machine. If this also fails, ULE simply picks the core with the lowest number of running threads on the machine.

ULE also balances threads periodically, every 500-1500ms (the duration of the period is chosen randomly). Unlike CFS, the periodic load balancing is performed only by core 0. Core 0 simply tries to even out the number of threads amongst the cores as follows: a thread from the most loaded core, the (*donor*), is migrated to the less loaded core, the (*receiver*). A core can only be a donor or a receiver once, and the load balancer iterates until no donor or receiver is found, meaning that a core may give away or receive at most one thread per load balancer invocation.

ULE also balances threads when the interactive and batch runqueues of a core are empty. ULE tries to steal from the most loaded core with which the idle core shares a cache. If ULE fails to steal a thread, it tries at a higher level of the topology and so on, until it finally manages to steal a thread. As with the periodic load balancer, the idle stealing mechanism steals at most one thread.

Periodic load balancing in ULE happens less often than in CFS, but more computation is involved in selecting a core during thread placement in ULE. The rationale is that having a better initial thread placement avoids the need for frequently running a global load balancer.[1]

3 Porting ULE to the Linux kernel

In this section we describe the problems encountered when porting ULE to Linux, and the main differences between our port and the original FreeBSD code.

The Linux kernel offers an API to add new schedulers to the kernel. Schedulers must implement the set of functions presented in Table 1. These functions are responsible for adding and removing threads in runqueues, picking threads to be scheduled, and placing threads on cores.

FreeBSD does not offer such an API to schedulers. Instead, it declares prototypes of the functions that must be defined, meaning that only one scheduler can be used at a time, as opposed to Linux, in which multiple scheduling classes can co-exist. Fortunately, functions inside the ULE scheduler can easily be mapped to their counterparts in Linux (see Table 1). In the few cases where the interfaces do not match, it was possible to find a workaround. For instance, Linux uses a single function to enqueue newly created threads and threads that have been woken up, while FreeBSD uses two functions. Linux uses a flag parameter in its function to distinguish between the two cases. It then suffices to use this flag to choose the corresponding FreeBSD function.

[1] In recent versions of FreeBSD, due to a bug, the periodic load balancer never executes [1]. In our ULE code we fixed the bug, and the load balancer is executed periodically.

Linux	FreeBSD equivalent	Usage
`enqueue_task`	`sched_add` for new threads `sched_wakeup` for woken up threads	Enqueue a thread in a runqueue
`dequeue_task`	sched_rem	Remove a thread from a runqueue
`yield_task`	sched_relinquish	Yield the CPU back to the scheduler
`pick_next_task`	sched_choose	Select the next task to be scheduled
`put_prev_task`	sched_switch	Update statistics about the task that just ran
`select_task_rq`	sched_pickcpu	Choose the CPU on which a new (or waking up) thread should be placed

Table 1: Linux scheduler API and equivalent functions in FreeBSD.

Other than interfaces, CFS and ULE also differ in some low-level assumptions. The most notable difference is related to the presence or absence of the current thread in the runqueue data structures. The Linux scheduling class mechanism relies on the assumption that the current thread stays in the runqueue data structure while it runs on a core. In ULE, a thread that runs on a core is removed from the runqueue data structure, and added back when its timeslice is depleted, so that the FIFO property holds. When trying to implement this behavior in Linux, we encountered several showstopper issues, such as kernel crashes for threads that tried to change their scheduling class. We decided instead to adhere to the Linux way of doing things, and leave the currently running thread in the runqueue. Because of that, we had to slightly change the ULE load balancing to avoid migrating a currently running thread.

Furthermore, in ULE, when migrating a thread from one CPU to another, the scheduler acquires the lock on both runqueues. In Linux, this locking mechanism lead to deadlocks. Therefore, we modified the ULE load balancing code to use the same mechanism as that of CFS.

Finally, in FreeBSD, ULE is responsible for scheduling all threads, whereas Linux uses different scheduling policies for different priority ranges (e.g., a realtime scheduler for high priority threads). In this work, we are mainly interested in comparing workloads with priorities falling in the CFS range (100-139). Hence, we scaled down the ULE penalty scores to fit within the CFS range.

4 Experimental environment

4.1 Machines

We evaluate ULE on a 32-core machine (AMD Opteron 6172) with 32GB of RAM. All experiments were performed on the latest LTS Linux kernel (4.9). We also ran experiments on a smaller desktop machine (8-core Intel i7-3770), reaching similar conclusions. Due to space limitations, we omit these results from the paper.

4.2 Workloads

To assess the performance of CFS and ULE, we used both synthetic benchmarks and realistic applications. Fibo is a synthetic application computing Fibonacci numbers. Hackbench [10] is a benchmark designed by the Linux community to stress the scheduler. It creates a large number of threads that run for a short amount of time and exchange data using pipes. We also selected 16 applications from the Phoronix test suite [2] based on their completion time. We excluded Phoronix applications that take more than 10 minutes to complete on a single core, or that were too short to allow reliable time measurements. The 16 Phoronix applications are: compilation benchmarks (build-apache, build-php), compression (7zip, gzip), image processing (c-ray, dcraw), scientific (himeno, hmmer, scimark), cryptography (john-the-ripper) and web (apache). We use the NAS benchmark suite [6] to benchmark HPC applications, the Parsec benchmark suite [7] to benchmark parallel applications, and Sysbench [3] with MySQL and RocksDB [16] as database benchmarks. We use a read-write workload for sysbench and RocksDB to schedule threads with different behaviors.

5 Evaluation of per-core scheduling

In this section, we run applications on a single core to avoid possible side effects introduced by the load balancer. The main difference between CFS and ULE in per-core scheduling is in the handling of batch threads: CFS tries to be fair to all threads, while ULE gives priority to interactive threads. We first analyze the impact of this design decision by co-scheduling a batch and an interactive application on the same core, and we show that under ULE batch applications can starve for an unbounded amount of time. We then show that starvation under ULE can occur even when the system is only running a single application. We conclude this section by comparing the performance of 37 applications, and show how different choices regarding the preemption of threads impact performance.

5.1 Fairness and starvation when co-scheduling applications

In this section, we analyse the behavior of CFS and ULE running a multi-application workload consisting of a compute-intensive thread that never sleeps (fibo, computing numbers), and an application whose threads mostly sleep (sysbench, a database, using 80 threads). Having more than 80 threads per core is not uncommon in datacenters [12]. These threads are never all active at the same time; they mostly wait for incoming requests, or for data stored on disk.

Fibo runs alone for 7 seconds, and then sysbench is launched. Both applications then run to completion. Figure 1(a) presents the runtime accumulated by fibo and sysbench on CFS, and Figure 1(b) presents the same quantities on ULE.

On CFS, sysbench completes in 235s, and then fibo runs alone. Both fibo and sysbench threads share the machine. When sysbench executes, the cumulative runtime of fibo progresses roughly half as fast as when it runs alone, meaning that fibo gets 50% of the core. This is expected: CFS tries to share the core fairly between the two applications. In practice, fibo gets a bit less than half of the CPU due to rounding errors.

On ULE, sysbench is able to complete the same workload in 143s, and then fibo runs by itself. fibo starves while sysbench is running. sysbench threads mainly sleep, so they are classified as interactive and get absolute priority over fibo. Since sysbench uses 80 threads, these threads are able to saturate a core, and prevent fibo from running. Figure 2 presents the evolution of the interactivity penalty of fibo and sysbench over time. Both applications start out as interactive (penalty of 0). The penalty of fibo quickly rises to the maximum value, and then fibo is no longer considered interactive. Sysbench threads, in contrast, remain interactive during their entire execution (penalty below the 30 limit). Thus, sysbench threads get absolute priority over the fibo thread. This situation persists as long as sysbench is running (i.e., the starvation time is not bounded by ULE).

Table 2 presents the total execution time of fibo and sysbench on CFS and ULE, and the latency of requests for sysbench. Sysbench runs 50% slower on CFS, because it shares the CPU with fibo, instead of running in isolation, as it does with ULE (290 transactions/s with CFS vs. 532 with ULE). Fibo is “stopped” during the execution of sysbench on ULE, but then gets to execute by itself, and thus can use the cache more efficiently than when running simultaneously with sysbench on CFS. Thus, fibo runs slightly faster on ULE than on CFS.

We found the strategy used by the ULE scheduler to work well with latency-sensitive applications. These applications are usually correctly classified as interactive and get priority over background threads. To achieve the same result in Linux, the latency-sensitive application would have to be executed by the realtime scheduler, which gets absolute priority over CFS.

	CFS	ULE
Fibo - Runtime	160s	158s
Sysbench - Transactions/s	290	532
Sysbench - Avg. latency	441ms	125ms

Table 2: Execution time of fibo and sysbench using CFS and ULE, average latency of requests in sysbench using CFS and ULE.

5.2 Fairness and starvation within a single application

The starvation exhibited by ULE in the multi-application scenario above also occurs in single-application workloads. We now exemplify this behavior using sysbench.

In ULE, newly created threads inherit the interactivity penalty of their parent at the time of the fork. In sysbench, the master thread is created with the interactivity penalty of the bash process from which it was forked. Since bash mostly sleeps, sysbench is created as an interactive process. The sysbench master thread initializes data and creates threads. While doing so, it never sleeps, and its interactivity penalty increases. The first threads are created with an interactivity penalty below the interactive threshold, while the remaining threads are created with an interactivity penalty above it. As a consequence, the first threads get absolute priority over the remaining ones. Since these threads spend most of their time waiting for new requests, their interactivity penalty stays low (it decreases to 0), and their priority remains higher than that of threads that were forked late in the initialization process. The latter threads sysbench may starve forever, if the interactive threads keep the CPU busy at all times.

Figure 3 presents the cumulative runtime of sysbench threads, and Figure 4 presents their interactivity penalty. Sysbench is configured to use 128 threads. The threads created early execute, and their interactivity penalty drops to 0. The threads created later never execute.

Counterintuitively, in this benchmark ULE actually performs better than CFS, because it avoids oversubscription: the machine runs as many threads as it can. As a consequence, ULE has a lower average latency than CFS. In general, we found that this starvation mechanism, seemingly problematic on paper, performs very well in applications where all threads compete to perform the same job.

In contrast to sysbench, the scientific applications we tested are not impacted by starvation, because their threads never sleep. After a short initialization period

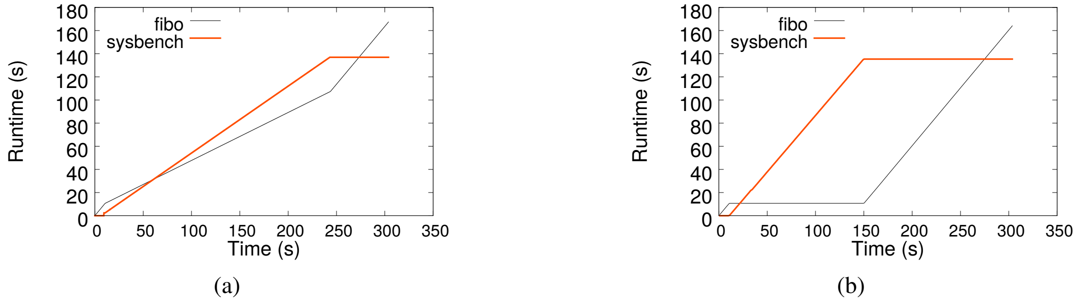


Figure 1: Cumulative runtime of fibo, and sysbench on (a) CFS, and (b) ULE. (a) On CFS, fibo continues to accumulate runtime, albeit more slowly, when sysbench executes, meaning that fibo is not starved. (b) On ULE, when sysbench executes, fibo stops accumulating runtime, meaning that it is starved.

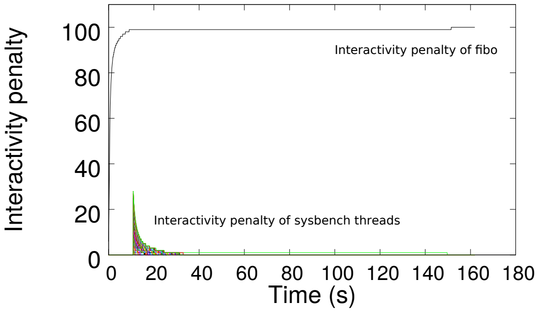


Figure 2: Interactivity penalty of threads over time. Fibo's penalty quickly rises to the maximum value, while the penalty of sysbench threads drops to 0.

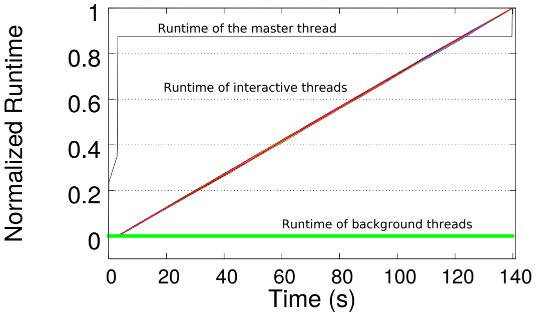


Figure 3: Cumulative runtime of threads of sysbench on ULE. The master thread first spawns 128 threads. 80 threads are classified as interactive and are executed, and 48 threads are classified as background and starve.

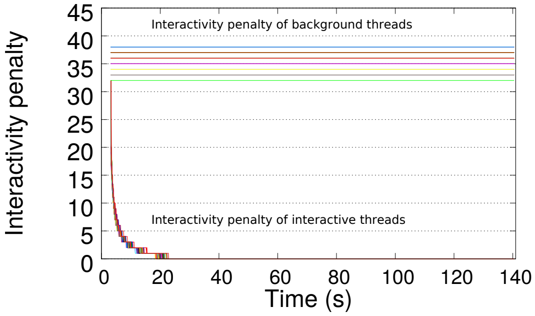


Figure 4: Interactivity penalty of the threads presented in Figure 3. Threads inherit the interactivity penalty of their parent when created. Some are created with a low penalty, and their penalty decreases as they execute (bottom of the graph), while other threads are created with a high penalty and never execute (top of the graph).

all threads are considered as background threads and are scheduled in a fair manner.

5.3 Performance analysis

We now analyze the impact of the per-core scheduling on the performance of 37 applications. We define "performance" as follows: for database workloads and NAS applications, we compare the number of operations per second, and for the other applications we compare "1/execution time". The higher the "performance", the better a scheduler performs. Figure 5 presents the performance difference between CFS and ULE on a single core, with percentages above 0 meaning that the application executes faster with ULE than CFS.

Overall, the scheduler has little influence on most workloads. Indeed, most applications use threads that all perform the same work, thus both CFS and ULE end up scheduling all of the threads in a round-robin fashion. The average performance difference is 1.5%, in favor of

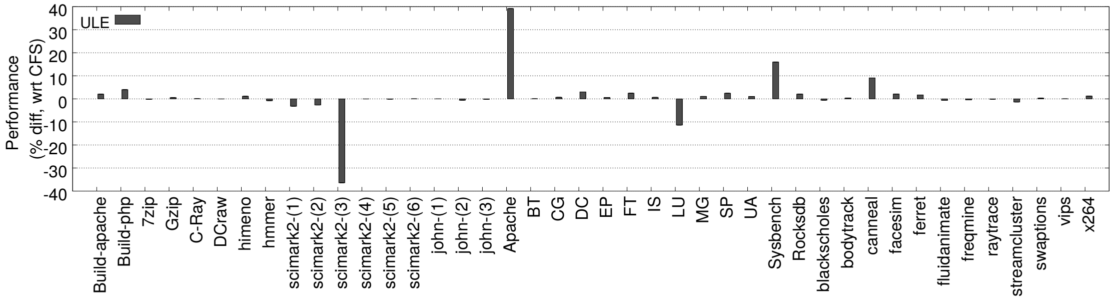


Figure 5: Performance of ULE with respect to CFS on a single core. A number higher than 0 means that the application runs faster on ULE than on CFS.

ULE. Still, scimark is 36% slower on ULE than CFS, and apache is 40% faster on ULE than CFS.

Scimark is a single-threaded Java application. It launches one compute thread, and the Java runtime executes other Java system threads in the background (for the garbage collector, I/O, etc.). When the application is executed with ULE, the compute thread can be delayed, because Java system threads are considered interactive and get priority over the computation thread.

The apache workload consists of two applications: the main server (*httpd*) running 100 threads, and *ab*, a single-threaded load injector. The performance difference between ULE and CFS is explained by different choices regarding thread preemption.

In ULE, full preemption is disabled, while CFS preempts the running thread when the thread that has just been woken up has a vruntime that is much smaller than the vruntime of the currently executing thread (1ms difference in practice). In CFS, *ab* is preempted 2 million times during the benchmark, while it never preempted with ULE. This behavior is explained as follows: *ab* starts by sending 100 requests to the httpd server, and then waits for the server to answer. When *ab* is woken up, it checks which requests have been processed and sends new requests to the server. Since *ab* is single-threaded, all requests sent to the server are sent sequentially. In ULE, *ab* is able to send as many new requests as it has received responses. In CFS, every request sent by *ab* wakes up a *httpd* thread, which preempts *ab*.

6 Evaluation of the load balancer

In this section, we analyze the impact of the load balancing and thread placement strategies on performance. In CFS and ULE, load balancing happens periodically, and thread placement occurs when threads are created or woken up. We first analyze the time it takes for the periodic load balancer to balance a static workload on all cores of the machine. We then analyze design choices made by CFS and ULE when placing threads. Next, we compare the performance of 37 applications running on CFS vs. ULE. Finally, we analyze the performance of multi-application workloads.

6.1 Periodic load balancing

CFS relies on a rather complex load metric. It uses a hierarchical load balancing strategy that runs every 4ms. ULE only tries to even out the number of threads on the cores. Load balancing happens less often (the period varies between 0.5s and 1.5s) and ignores the hardware topology. We now evaluate how these strategies impact the time needed to balance the load on the machine.

To that end, we pin 512 spinning threads on core 0, we launch a taskset command to unpin the threads, and we let the load balancer balance the load between cores. All threads perform the same work (an infinite empty loop), so we expect the load balancer to place 16 threads on each of the 32 cores. Figure 6 presents the evolution over time of the number of threads per core. In the figure, each of the 32 lines represents the number of threads on a given core. The taskset command that unpins threads is launched at 14.5s.

On ULE, as soon as the threads are unpinned, idle cores steal threads (at most one per core) from core 0, thus right after the unpinning, core 0 has $512-31=481$ threads while every other core has 1 thread. Over time, the periodic load balancer is triggered and tries to balance the thread count. However, as the load balancer only migrates one thread at a time from core 0, it takes more that 450 load balancer invocations or about 240 seconds to reach a balanced state.

CFS balances the load much faster. 0.2 seconds after the unpinning, CFS has migrated more that 380 threads from core 0. Surprisingly, CFS never achieves perfect load balance. CFS only balances the load between NUMA nodes when the imbalance between the two nodes is “big enough” (25% load difference in practice). So cores in one node can have 18 threads while cores in another only have 15.

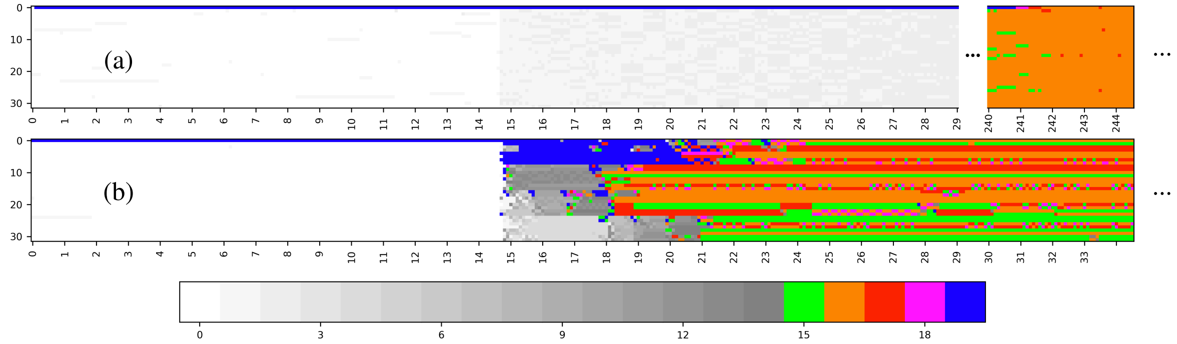


Figure 6: Number of threads per core over time on (a) ULE and (b) CFS. Each line represents a core (32 in total), time passes on the x-axis (in seconds), and colors represent the numbers of threads on the core. Thread counts below 15 are represented in shades of grey. Threads are pinned on core 0 for the first 14.5 seconds of the execution.

While the load balancing strategy used by CFS is well suited for solving a large imbalance loads in the system, it is less suited when a perfect load balance is important for performance.

6.2 Thread placement

We study placement of threads using c-ray, an image processing application from the phoronix benchmark suite. C-ray starts by creating 512 threads. Threads are not pinned at creation time, so the scheduler chooses a core for each thread. Then all threads wait on a barrier before performing the computation. Since all threads behave in the same way, we would expect ULE to perform better than CFS in that configuration: ULE always forks threads on the core with the lowest number of threads, so the load should be perfectly balanced from the start.

Figure 7 presents the evolution in the number of runnable threads per core over time. Load is always balanced in ULE, but surprisingly it takes more than 11 seconds for ULE to have all threads runnable, while it only takes 2 seconds for CFS. This delay is explained by starvation. C-ray uses a cascading barrier in which thread 0 wakes up thread 1, thread 1 wakes up thread 2, etc. Threads are originally created with different interactivity penalties, and some threads are initially interactive, while others are initially batch (same reason as in sysbench, see Section 5.2). When a batch thread is woken up, it might starve until all interactive threads are done, or until their penalty has increased enough for them to be downgraded to the batch runqueue. In practice, in c-ray, threads never sleep after the barrier, so eventually all threads become batch, but, before they do, threads that were initially categorized as batch cannot wake up other threads. Thus, it takes 11 seconds for all threads to be woken up after the barrier.

CFS on the contrary is fair, and all threads are quickly woken up. Then, CFS runs into the imperfect load balancing issue that we explained in Section 6.1.

Despite these load balancing differences, c-ray completes in the same time on CFS and ULE, because c-ray creates more threads than cores, and because both schedulers always keep all cores busy. Preemptions do occur more often with CFS, but do not affect the performance.

6.3 Performance analysis

Figure 8 presents the performance difference between CFS and ULE in a multicore context. The average performance difference between CFS and ULE is small: 2.75% in favor of ULE.

MG, from the NAS benchmark suite, benefits the most from ULE's load balancing strategy: it is 73% faster on ULE than on CFS. MG spawns as many threads as there are cores in the machine, and all threads perform the same computations. When a thread has finished its computation, it waits on a spin-barrier for 100ms and then sleeps if some threads are still computing. ULE correctly places one thread per core, and then never migrates them again. Threads spend very little time waiting for each other in the barriers, and never sleep. In contrast, CFS reacts to micro changes in the load of cores (e.g., due to a kernel thread waking up), and sometimes wrongly places two MG threads on the same core. Since MG uses barriers, the two threads scheduled on the same core end up delaying the whole application. The delay is more than 50% because threads scheduled alone on their cores go to sleep, and then have to be woken up, thus adding latency to the barriers. This suboptimal thread placement also explains the performance difference between CFS and ULE on FT and UA. The simple approach of balancing the number of threads used by ULE works better on HPC-like applications because it ends up placing one thread per core and then never migrates them again.

Sysbench, is slower on ULE due to the overhead of the ULE load balancer. When a thread wakes up, ULE scans

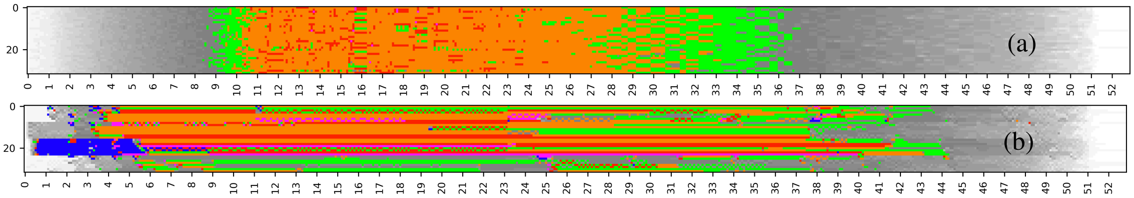


Figure 7: Number of threads per core over time on c-ray on (a) ULE and (b) CFS. Contrary to Figure 6, threads do not start pinned on core 0.

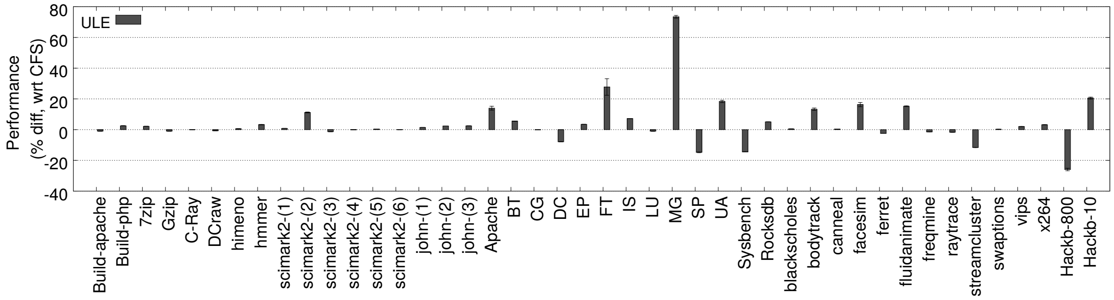


Figure 8: Performance of ULE with respect to CFS on a multicore.

the cores of the machine to find an appropriate core for the thread, and, at worst, may scan all cores three times. This worst case scenario happens on most wakeups in sysbench, resulting in 13% of all CPU cycles being spent on scanning cores. To validate this assumption, we replaced the ULE wakeup function by a simple one that returns the CPU on which the thread was previously running, and then observed no difference between ULE and CFS.

In all the benchmarks we tested, 13% is the highest time spent in the scheduler we observed in ULE, and 2.6% is the highest time spent in the scheduler we observed in CFS. Note that ULE runs into a corner case situation with sysbench, but has a low overhead on other benchmarks, even when they spawn a large number of threads: for instance in hackbench (32 000 threads), the overhead of ULE is 1% (compared to 0.3% for CFS).

6.4 Multi application workloads

Finally, we evaluate the combination of interactive and background workloads using a set of different applications: c-ray + EP (from NAS) is a workload where both applications are considered background by ULE, fibo + sysbench and blackscholes + ferret are workloads where only one application is interactive, and apache + sysbench is a fully interactive workload. Figure 9 shows the performance of CFS and ULE with respect to the performance of the application running alone on the machine (higher is better). Overall, most applications run slower when they are co-scheduled with another application.

When both applications are non-interactive (c-ray + EP), CFS and ULE perform similarly. This is expected, as they schedule background threads in a similar way. EP runs slightly faster on ULE when executed alone, and this performance difference is still present when it is co-scheduled with c-ray. When both applications are interactive, CFS and ULE also perform similarly.

For blackscholes + ferret, ULE gives priority to the interactive application, and ferret is not impacted by being co-scheduled with blackscholes. Blackscholes however runs more than 80% slower. In that context, blackscholes does not fully starve because ferret does not use 100% of all cores. CFS on the contrary shares the CPU fairly, and both applications suffer equally (the impact of co-scheduling on these applications is less than 50% because neither ferret nor blackscholes scales to 32 cores).

Surprisingly when co-scheduled with fibo, sysbench performs worse on ULE than on CFS even though it is correctly categorized as interactive and gets priority over fibo threads. The lack of preemption in ULE explains the performance difference. MySQL does not achieve perfect scaling and, when executed on 32 cores, lock contention forces the threads to sleep when waiting for the locks to be released. Thus, fibo does not starve. When a MySQL lock is released, ULE does not preempt the currently running thread (usually fibo) to schedule a new MySQL thread to enter the critical section. This adds delays (of up to the length of fibo's timeslice, between 7.8ms and 78ms) to the execution of sysbench.

7 Related work

Previous works have compared the design choices made by FreeBSD and Linux. Abaffy et al. [4, 5] compare the

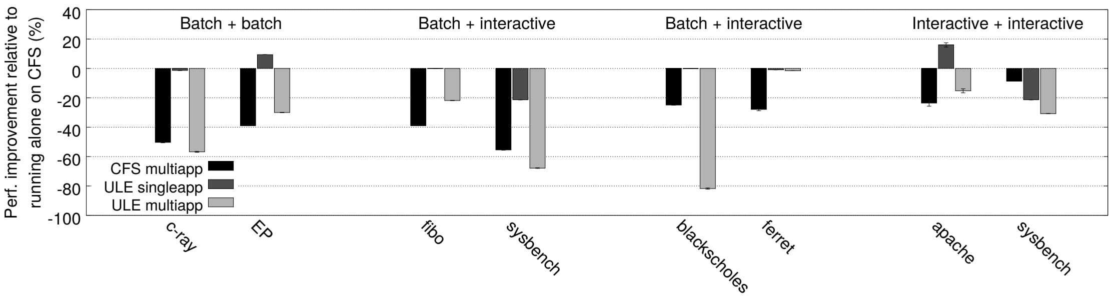


Figure 9: Performance of CFS and ULE on multi application workloads with respect to the performance of the application running alone on CFS.

average waiting time of threads in scheduler runqueues. Schneider et al. [17] compare the networking stack performance of the two operating systems. Design choices made by FreeBSD are also frequently discussed on the Linux kernel mailing list [20]. This study differs in its approach: instead of comparing two complete operating systems, we ported the FreeBSD ULE scheduler to Linux. To the best of our knowledge, this is the first apples-to-apples comparison of the design of ULE and CFS.

The Linux scheduler design has also been discussed in previous works. Torrey et al. [19] compare the latency of the Linux scheduler against a custom implementation of a multilevel feedback queue. Wong et al. compare the fairness of CFS with the O(1) scheduler that was the default Linux scheduler prior to 2.6.23 [23], and with a RSDL scheduler (Rotating Staircase Deadline Scheduler) [22]. Groves et al. [9] compare the overhead of CFS against BFS (Brain Fuck Scheduler), a simplistic scheduler aimed at improving responsiveness on machines with few cores. Other work has studied the overhead of schedulers. Kanev et al. [12] report that the CFS alone accounts for more than 5% of all datacenter cycles.

The performance of operating systems is frequently assessed by measuring the evolution of performance between kernel versions. The Linux Kernel Performance project [8] started in 2005 to measure performance regressions in the Linux Kernel. Mollison et al. [14] propose Litmus tests to find performance regressions in schedulers. Performance issues in operating systems are also frequently reported in the Systems community. Lozi et al. [13] report bugs in the Linux scheduler that could lead to cores being permanently left idle while work was waiting to be scheduled on other cores. Harji et al. [11] report similar performance bugs in earlier kernel versions. During the work on this paper we also reported bugs in the scheduler to the FreeBSD community [1]. In this study we chose to compare "glitch free" versions of ULE and CFS by fixing obvious bugs that were not intended as features of the schedulers.

8 Conclusion

Scheduling threads on a multicore machine is hard. In this paper, we perform a fair comparison of the design choices of two widely used schedulers: the ULE scheduler from FreeBSD and CFS from Linux. We show that they behave differently even on simple workloads, and that no scheduler performs better than the other on all workloads.

References

[1] [PATCH] Fix bug in which the long term ULE load balancer is executed only once. `https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=223914`.

[2] Phoronix Test Suite. `https://www.phoronix.com`.

[3] Sysbench. `https://github.com/akopytov/sysbench`.

[4] Abaffy, J., and Krajčovič, T. Latencies in Linux and FreeBSD kernels with different schedulers-O(1), CFS, 4BSD, ULE. In *Proceedings of the 2nd International Multi-Conference on Engineering and Technological Innovation* (2009), pp. 110–115.

[5] Abaffy, J., and Krajčovič, T. Pi-ping-benchmark tool for testing latencies and throughput in operating systems. *Innovations in Computing Sciences and Software Engineering* (2010), 557–560.

[6] Bailey, D., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H., Venkatakrishnan, V., and Weeratunga, S. The NAS parallel benchmarks summary and pre-

liminary results. In *Supercomputing '91.* (Nov. 1991), pp. 158–165.

[7] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT* (2008).

[8] Chen, T., Ananiev, L. I., and Tikhonov, A. V. Keeping kernel performance from regressions. In *Linux Symposium* (2007), vol. 1, pp. 93–102.

[9] Groves, T., Knockel, J., and Schulte, E. BFS vs. CFS scheduler comparison. *The University of New Mexico* (2009). http://cs. unm. edu/˜eschulte/classes/cs587/data/bfsv-cfs_groves-knockel-schulte. pdf (accessed Jan. 5, 2017).

[10] Hackbench. `https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git/tree/src/hackbench?h=v0.93`, 2008.

[11] Harji, A. S., Buhr, P. A., and Brecht, T. Our troubles with Linux and why you should care. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (New York, NY, USA, 2011), APSys '11, pp. 2:1–2:5.

[12] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., and Brooks, D. Profiling a warehouse-scale computer. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on* (2015), IEEE, pp. 158–169.

[13] Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V., and Fedorova, A. The Linux scheduler: a decade of wasted cores. In *EuroSys'16* (2016), ACM, pp. 1:1–1:16.

[14] Mollison, M. S., Brandenburg, B., and Anderson, J. H. Towards unit testing real-time schedulers in LITMUS^{RT}. In *Proceedings of the 5th Workshop on Operating Systems Platforms for Embedded Real-Time Applications* (2009), OSPERT.

[15] Roberson, J. ULE: a modern scheduler for FreeBSD.

[16] RocksDB - Performance Benchmarks. `https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks`.

[17] Schneider, F., and Wallerich, J. Performance evaluation of packet capturing systems for high-speed networks. In *Proceedings of the 2005 ACM conference on Emerging network experiment and technology* (2005), ACM, pp. 284–285.

[18] The design of CFS. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`.

[19] Torrey, L. A., Coleman, J., and Miller, B. P. A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler. *Software: Practice and Experience 37*, 4 (2007), 347–364.

[20] Torvalds, L. Is sendfile all that sexy? `http://yarchive.net/comp/linux/sendfile.html`.

[21] ULE port in Linux. `https://github.com/JBouron/linux/tree/loadbalancing`.

[22] Wang, S., Chen, Y., Jiang, W., Li, P., Dai, T., and Cui, Y. Fairness and interactivity of three CPU schedulers in Linux. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA'09. 15th IEEE International Conference on* (2009), IEEE, pp. 172–177.

[23] Wong, C., Tan, I., Kumari, R., Lam, J., and Fun, W. Fairness and interactive performance of O(1) and CFS Linux kernel schedulers. In *Information Technology, 2008. ITSim 2008. International Symposium on* (2008), vol. 4, IEEE, pp. 1–8.

The Design and Implementation of Hyperupcalls

Nadav Amit
VMware Research

Michael Wei
VMware Research

Abstract

The virtual machine abstraction provides a wide variety of benefits which have undeniably enabled cloud computing. Virtual machines, however, are a double-edged sword as hypervisors they run on top of must treat them as a black box, limiting the information which the hypervisor and virtual machine may exchange, a problem known as the *semantic gap*. In this paper, we present the design and implementation of a new mechanism, hyperupcalls, which enables a hypervisor to safely execute verified code provided by a guest virtual machine in order to transfer information. Hyperupcalls are written in C and have complete access to guest data structures such as page tables. We provide a complete framework which makes it easy to access familiar kernel functions from within a hyperupcall. Compared to state-of-the-art paravirtualization techniques and virtual machine introspection, Hyperupcalls are much more flexible and less intrusive. We demonstrate that hyperupcalls can not only be used to improve guest performance for certain operations by up to 2× but hyperupcalls can also serve as a powerful debugging and security tool.

1 Introduction

Hardware virtualization introduced the abstraction of a virtual machine (VM), enabling hosts known as *hypervisors* to run multiple operating systems (OSs) known as *guests* simultaneously, each under the illusion that they are running in their own physical machine. This is achieved by exposing a hardware interface which mimics that of true, physical hardware. The introduction of this simple abstraction has led to the rise of the modern data center and the cloud as we know it today. Unfortunately, virtualization is not without drawbacks. Although the goal of virtualization is for VMs and hypervisors to be oblivious from each other, this separation renders both sides unable to understand decisions made on the other side, a problem known as the *semantic gap*.

		Paravirtual, Executed by:		**Uncoordinated**
		Hypervisor	Guest	Introspection
Requestor	Guest	Hypercalls	Pre-Virt [42]	HVI [72]
	HV	**Hyperupcalls**	Upcalls	VMI [25]

Table 1: *Hypervisor-Guest Communication Mechanisms.* Hypervisors (HV) and guests may communicate through a variety of mechanisms, which are characterized by who initiates the communication, who executes and whether the channel for communication is coordinated (paravirtual). Shaded cells represent channels which require context switches.

Addressing the semantic gap is critical for performance. Without information about decisions made in guests, hypervisors may suboptimally allocate resources. For example, the hypervisor cannot know what memory is free in guests without understanding their internal OS state, breaking the VM abstraction. State-of-the-art hypervisors today typically bridge the semantic gap with *paravirtualization* [11, 58], which makes the guest aware of the hypervisor. Paravirtualization alleviates the guest from the limitations of the physical hardware interface and allows direct information exchange with the hypervisor, improving overall performance by enabling the hypervisor to make better resource allocation decisions.

Paravirtualization, however, involves the execution of code both in the context of the hypervisor and the guest. *Hypercalls* require that the guest make a request to be executed in the hypervisor, much like a system call, and *upcalls* require that the hypervisor make a request to be executed in the guest. This design introduces a number of drawbacks. First, paravirtual mechanisms introduce context switches between hypervisors and guests, which may be substantial if frequent interactions between guests and the hypervisor are needed [7]. Second, the requestor of a paravirtual mechanism must wait for it to be serviced in another context which may be busy, or waking the guest if it is idle. Finally, paravirtual mechanisms couple the design of the hypervisor and guest: paravirtual mechanisms need to be implemented for each guest and hypervisor, increasing complexity [46] and hampering maintainability [77]. Adding paravirtual features requires updating both the guest and hypervisor with a new interface [69] and has the potential to introduce bugs and the attack surface [47, 75].

A different class of techniques, VM introspection (VMI) [25] and the reverse, hypervisor introspection (HVI) [72] aim to address some of the shortcomings of paravirtualization by *introspecting* the other context, enabling communication transfer without context switching or prior coordination. These techniques however, are fragile: small changes in data structures, behavior or even security hardening [31] can break introspective mechanisms, or worse, introduce security vulnerabilities. As a result, introspection is usually relegated to the area of intrusion detection systems (IDSs) which detect malware or misbehaving applications.

In this paper, we describe the design and implementation of hyperupcalls [1], a technique which enables a hypervisor to communicate with a guest, like an upcall, but without a context switch, like VMI. This is achieved through the use of verified code, which enables a guest to communicate to the hypervisor in a flexible manner while ensuring that the guest cannot provide misbehaving or malicious code. Once a guest registers a hyperupcall, the hypervisor can execute it to perform actions such as locating free guest pages or running guest interrupt handlers without switching into the guest.

Hyperupcalls are easy to build: they are written in a high level language such as C, and we provide a framework which allows hyperupcalls to share the same codebase and build system as the Linux kernel that may be generalized to other operating systems. When the kernel is compiled, a toolchain translates the hyperupcall into verifiable bytecode. This enables hyperupcalls to be easily maintained. Upon boot, the guest registers the hyperupcalls with the hypervisor, which verifies the bytecode and compiles it back into native code for performance. Once recompiled, the hypervisor may invoke the hyperupcall at any time.

We show that using a hyperupcalls can significantly improve performance by allowing a hypervisor to be proactive about resource allocation, rather than waiting for guests to react through existing mechanisms. We build hyperupcalls for memory reclamation and dealing with interprocessor interrupts (IPIs) and show a performance improvement of up to 2×. In addition to improving performance, hyperupcalls can also enhance both the security and debuggability of systems in virtual environments. We develop a hyperupcall to enables guests to write-protect memory pages without the use of specialized hardware, and another which enables `ftrace` [57] to capture both guest and hypervisor events in a unified trace, allowing us to gain new insights on performance in virtualized environments.

This paper makes the following contributions:

- We build a taxonomy of mechanisms for bridging the semantic gap between hypervisor and guests and place hyperupcalls within that taxonomy (§2).
- We describe and implement hyperupcalls (§3) with:
  - An environment for writing hyperupcalls and a framework for using guest code (§3.1)
  - A compiler (§3.2) and verifier (§3.4) for hyperupcalls which addresses the complexities and limitations of verified code.
  - Registration (§3.3) and execution (§3.5) mechanisms for hyperupcalls.
- We prototype and evaluate hyperupcalls and show that hyperupcalls can improve performance (§4.3, §4.2), security (§4.5) and debuggability (§4.4).

2 Communication Mechanisms

It is now widely accepted that in order to extract the most performance and utility from virtualization, hypervisors and their guests need to be aware of one another. To that end, a number of mechanisms exist to facilitate communication between hypervisors and guests. Table 1 summarizes these mechanisms, which can be broadly characterized by the requestor, the executor, and whether the mechanism requires that the hypervisor and the guest coordinate ahead of time.

In the next section, we discuss these mechanisms and describe how hyperupcalls fulfill a need for a communication mechanism where the hypervisor makes and executes its own requests without context switching. We begin by introducing state-of-the-art paravirtual mechanisms in use today.

2.1 Paravirtualization

Hypercalls and upcalls. Most hypervisors today leverage paravirtualization to communicate across the semantic gap. Two mechanisms in widespread use today are *hypercalls*, which allow guests to invoke services provided by the hypervisor, and *upcalls*, which enable the hypervisor to make requests to guests. Paravirtualization means that the interface for these mechanisms are coordinated ahead of time between hypervisor and guest [11].

One of the main drawbacks of upcalls and hypercalls is that they require a context switch as both mechanisms are executed on the opposite side of the request. As a result, these mechanisms must be invoked with care. Invoking

[1] Hyperupcalls were previously published as “hypercallbacks” [5].

a hypercall or upcall too frequently can result in high latencies and computing resource waste [3].

Another drawback of upcalls in particular that the requests are handled by the guest, which could be busy handling other tasks. If the guest is busy or if a guest is idle, upcalls incur the additional penalty of waiting for the guest to be free or for the guest or woken up. This can take an unbounded amount of time, and hypervisors may have to rely on a penalty system to ensure guests respond in a reasonable amount of time.

Finally, by increasing the coupling between the hypervisor and its guests, paravirtual mechanisms can be difficult to maintain over time. Each hypervisor have their own paravirtual interfaces, and each guest must implement the interface of each hypervisor. The paravirtual interface is not thin: Microsoft's paravirtual interface specification is almost 300 pages long [46]. Linux provides a variety of paravirtual hooks, which hypervisors can use to communicate with the VM [78]. Despite the effort to standardize the paravirtualization interfaces they are incompatible with each other, and evolve over time, adding features or even removing some (e.g., Microsoft hypervisor event tracing). As a result, most hypervisors do not fully support efforts to standardize interfaces and specialized OSs look for alternative solutions [45, 54].

Pre-virtualization. Pre-virtualization [42] is another mechanism in which the guest requests services from the hypervisor, but the requests are served in the context of the guest itself. This is achieved by code injection: the guest leaves stubs, which the hypervisor fills with hypervisor code. Pre-virtualization offers an improvement over hypercalls, as they provide more flexible interface between the guest and the hypervisor. Arguably, pre-virtualization suffers from a fundamental limitation: code that runs in the guest is deprivileged and cannot perform sensitive operations, for example, accessing shared I/O devices. As a result, in pre-virtualization, the hypervisor code that runs in the guest still needs to communicate with the privileged hypervisor code using hypercalls.

2.2 Introspection

Introspection occurs when a hypervisor or guest attempts to infer information from the other context without directly communicating with it. With introspection, no interface or coordination is required. For instance, a hypervisor may attempt to infer the state of completely unknown guests simply by their memory access patterns. Another difference between introspection and paravirtualization is that no context switch occurs: all the code to perform introspection is executed in the requestor.

Virtual machine introspection (VMI). When a hypervisor introspects a guest, it is known as VMI [25]. VMI was first introduced to enhance VM security by providing intrusion detection (IDS) and kernel integrity checks from a privileged host [10, 24, 25]. VMI has also been applied to checkpointing and deduplicating VM state [1], as well as monitoring and enforcing hypervisor policies [55]. These mechanisms range from simply observing a VM's memory and I/O access patterns [36] to accessing VM OS data structures [16], and at the extreme end they may modify VM state and even directly inject processes into it [26, 19]. The primary benefits of VMI are that the hypervisor can directly invoke VMI without a context switch, and the guest does not need to be "aware" that it is inspected for VMI to function. However, VMI is fragile: an innocuous change in the VM OS, such as a hotfix which adds an additional field to a data structure could render VMI non-functional [8]. As a result, VMI tends to be a "best effort" mechanism.

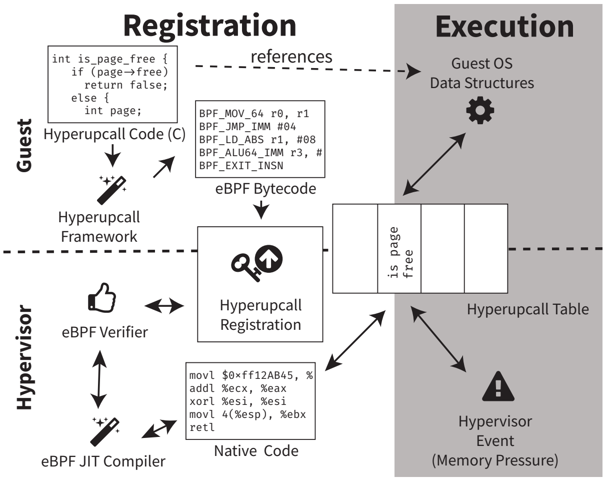
HVI. Used to a lesser extent, a guest may introspect the hypervisor it is running on, known as hypervisor introspection (HVI) [72, 61]. HVI is typically employed either to secure a VM from untrusted hypervisors [62] or by malware to circumvent hypervisor security [59, 48].

2.3 Extensible OSes

While hypervisors provide a fixed interface, OS research suggested along the years that flexible OS interfaces can improve performance without sacrificing security. The Exokernel provided low level primitives, and allowed applications to implement high-level abstractions, for example for memory management [22]. SPIN allowed to extend kernel functionality to provide application-specific services, such as specialized interprocess communication [13]. The key feature that enables these extensions to perform well without compromising security, is the use of a simple byte-code to express application needs, and running this code at the same protection ring as the kernel. Our work is inspired by these studies, and we aim to design a flexible interface between the hypervisor and guests to bridge the semantic gap.

2.4 Hyperupcalls

This paper introduces hyperupcalls, which fulfill a need for a mechanism for the hypervisor to communicate to the guest which is coordinated (unlike VMI), executed by the hypervisor itself (unlike upcalls) and does not require context switches (unlike hypercalls). With hyperupcalls, the VM coordinates with the hypervisor by registering verifiable code. This code is then executed by the hypervisor in response to events (such as memory pressure, or



```
int is_page_free {
   if (page→free)
     return false;
   else {
     int page;
```

```
BPF_MOV_64 r0, r1
BPF_JMP_IMM #04
BPF_LD_ABS r1, #08
BPF_ALU64_IMM r3, #
BPF_EXIT_INSN
```

```
movl $0×ff12AB45, %
addl %ecx, %eax
xorl %esi, %esi
movl 4(%esp), %ebx
retl
```

Figure 1: *System Architecture.* Hyperupcall registration (left) consists of compiling C code, which may reference guest data structures, into verifiable bytecode. The guest registers the generated bytecode with the hypervisor, which verifies its safety, compiles it into native code and sets it in the VM hyperupcall table. When the hypervisor encounters an event (right), such as a memory pressure, it executes the respective hyperupcall, which can access and update data structures of the guest.

VM entry/exit). In a way, hyperupcalls can be thought of as upcalls executed by the hypervisor.

In contrast to VMI, the code to access VM state is provided by the guest so the hyperupcalls are fully aware of guest internal data structures— in fact, hyperupcalls are built with the guest OS codebase and share the same code, thereby simplifying maintenance while providing the OS with an expressive mechanism to describe its state to underlying hypervisors.

Compared to upcalls, where the hypervisor makes asynchronous requests to the guest, the hypervisor can execute a hyperupcall at any time, even when the guest is not running. With an upcall, the hypervisor is at the mercy of the guest, which may delay the upcall [6]. Furthermore, because upcalls operate like remote requests, upcalls may be forced to implement OS functionality in a different manner. For example, when flushing remote pages in memory ballooning [71], the canonical technique for identifying free guest memory, the guest increases memory pressure using a dummy process to free pages. With a hyperupcall, the hypervisor can act as if it were a guest kernel thread and scan the guest for free pages directly.

Hyperupcalls resemble pre-virtualization, in that code is transferred across the semantic gap. Transferring code not only allows for more expressive communication, but it also moves the execution of the request to the other side of the gap, enhancing performance and functionality. Unlike pre-virtualization, the hypervisor cannot trust the code being provided by the virtual machine, and the hypervisor must ensure that execution environment for the hyperupcall is consistent across invocations.

	Local Hyperupcalls	**Global Hyperupcalls**
event examples	VM-exit VM-entry interrupt injection page mapping	memory reclaim memory aging VCPU preemption
use	notifications and local policy decisions	global policy decisions
preemptable	yes	no
memory mappings	host user-space	host kernel-space
memory limit	high	low
memory pinning	no	yes
callback chaining	yes	no
hyperupcall examples	IPI handling security agent tracing	scheduler activation memory discard hints

Table 2: *Hyperupcall event types.* The hypervisor enforces certain limitations on global hyperupcalls, which are used to make policy decisions.

3 Architecture

Hyperupcalls are short verifiable programs provided by guests to the hypervisor to improve performance or provide additional functionality. Guests provide hyperupcalls to the hypervisor through a *registration* process at boot, allowing the hypervisor to access the guest OS state and provide services by *executing* them after verification. The hypervisor runs hyperupcalls in response to events or when it needs to query guest state. The architecture of hyperupcalls and the system we have built for utilizing them is depicted in Figure 1.

We aim to make hyperupcalls as simple as possible to build. To that end, we provide a complete *framework* which allows a programmer to write hyperupcalls using the guest OS codebase. This greatly simplifies the development and maintenance of hyperupcalls. The framework compiles this code into verifiable code which the guest registers with the hypervisor. In the next section, we describe how an OS developer writes a hyperupcall using our framework.

3.1 Building Hyperupcalls

Guest OS developers write hyperupcalls for each hypervisor event they wish to handle. Hypervisors and guests agree on these events, for example VM entry/exit, page mapping or virtual CPU (VCPU) preemption. Each hyperupcall is identified by a predefined identifier, much like the UNIX system call interface [56]. Table 2 gives examples of events a hyperupcall may handle.

3.1.1 Providing Safe Code

One of the key properties of hyperupcalls is that the code must be guaranteed to not compromise the hypervisor. In order for a hyperupcall to be safe, it must only be able to access a restricted memory region dictated by the hypervisor, run for a limited period of time without blocking, sleeping or taking locks, and only use hypervisor services that are explicitly permitted.

Since the guest is untrusted, hypervisors must rely on a security mechanism which guarantees these safety properties. There are many solutions that we could have chosen: software fault isolation (SFI) [70], proof-carrying code [51] or safe languages such as Rust. To implement hyperupcalls, we chose the enhanced Berkeley Packet Filter (eBPF) VM.

We chose eBPF for several reasons. First, eBPF is relatively mature: BPF was introduced over 20 years ago and is used extensively throughout the Linux kernel, originally for packet filtering but extended to support additional use cases such as sandboxing system calls (`seccomp`) and tracing of kernel events [34]. eBPF enjoys wide adoption and is supported by various runtimes [14, 49]. Second, eBPF can be provably verified to have the safety properties we require, and Linux ships with a verifier and JIT which verifies and efficiently executes eBPF code [74]. Finally, eBPF has a LLVM compiler backend, which enables eBPF bytecode to be generated from a high level language using a compiler frontend (Clang). Since OSes are typically written in C, the eBPF LLVM backend provides us with a straightforward mechanism to convert unsafe guest OS source code into verifiably safe eBPF bytecode.

3.1.2 From C to eBPF — the *Framework*

Unfortunately, writing a hyperupcall is not as simple recompiling OS code into eBPF bytecode. However, our framework aims to make the process of writing a hyperupcalls simple and maintainable as possible. The framework provides three key features that simplify the writing of hyperupcalls. First, the framework takes care of dealing with guest address translation issues so guest OS symbols are available to the hyperupcall. Second, the framework addresses limitations of eBPF, which places significant constraints on C code. Finally, the framework defines a simple interface which provides the hyperupcall with data so it can execute efficiently and safely.

Guest OS symbols and memory. Even though hyperupcalls have access to the entire physical memory of the guest, accessing guest OS data structures requires knowing where they reside. OSes commonly use kernel address space layout randomization (KASLR) to randomize the virtual offsets for OS symbols, rendering them unknown during compilation time. Our framework enables OS symbol offsets to be resolved at runtime by associating pointers using address space attributes and injecting code to adjust the pointers. When a hyperupcall is registered, the guest provides the actual symbol offsets enabling a hyperupcall developer to reference OS symbols (variables and data structures) in C code as if they were accessed by a kernel thread.

Global / Local Hyperupcalls. Not all hyperupcalls need to be executed in a timely manner. For example, notifications informing the guest of hypervisor events such as a VM-entry/exit or interrupt injection only affect the guest and not the hypervisor. We refer to hyperupcalls that only affect the guest that registered it as local, and hyperupcalls that affect the hypervisor as a whole as global. If a hyperupcall is registered as local, we relax the timing requirement and allow the hyperupcall to block and sleep. Local hyperupcalls are accounted in the VCPU time of the guest similar to a trap, so a misbehaving hyperupcall penalizes itself.

Global hyperupcalls, however, must complete their execution in a timely manner. We ensure that for the guest OS pages requested by global hyperupcalls are pinned during the hyperupcall, and restrict the memory that can be accessed to 2% (configurable) of the guest's total physical memory. Since local hyperupcalls may block, the memory they use does not need to be pinned, allowing local hyperupcalls to address all of guest memory.

Addressing eBPF limitations. While eBPF is expressive, the safety guarantees of eBPF bytecode mean that it is not Turing-complete and limited, so only a subset of C code can be compiled into eBPF. The major limitations of eBPF are that it does not support loops, the ISA does not contain atomics, cannot use self-modifying code, function pointers, static variables, native assembly code, and cannot be too long and complex to be verified.

One of the consequences of these limitations is that hyperupcall developers must be aware of the code complexity of the hyperupcall, as complex code will fail the verifier. While this may appear to be an unintuitive restriction, other Linux developers using BPF face the same restriction, and we provide a helper functions in our framework to reduce complexity, such as `memset` and `memcpy`, as well as functions that perform native atomic

Helper Name	Function
send_vcpu_ipi	Send an interrupt to VCPU
get_vcpu_register	Read a VCPU register
set_vcpu_register	Read a VCPU register
memcpy	memcpy helper function
memset	memset helper function
cmpxchg	compare-and-swap
flush_tlb_vcpu	Flush VCPU's TLB
get_exit_info	Get info on an VM_EXIT event

Table 3: *Selected hyperupcall helper functions.* The hyperupcall may call these functions implemented in the hypervisor, as they cannot be verified using eBPF.

operations such as cmpxchg. A selection of these helper functions is shown in Table 3. In addition, our framework masks memory accesses (§3.4), which greatly reduces the complexity of verification. In practice, as long as we were careful to unroll loops, we did not encounter verifier issues while developing the use cases in (§4) using a setting of 4096 instructions and a stack depth of 1024.

Hyperupcall interface. When a hypervisor invokes a hyperupcall, it populates a context data structure, shown in Table 4. The hyperupcall receives an event data structure which indicates the reason the callback was called, and a pointer to the guest (in the address space of the hypervisor, which is executing the hyperupcall). When the hyperupcall completes, it may return a value, which can be used by the hypervisor.

Writing the hyperupcall. With our framework, OS developers write C code which can access OS variables and data structures, assisted by the helper functions of the framework. A typical hyperupcall will read the event field, read or update OS data structures and potentially return data to the hypervisor. Since the hyperupcall is part of the OS, the developers can reference the same data structures used by the OS itself—for example, through header files. This greatly increases the maintainability of hyperupcalls, since data layout changes are synchronized between the OS source and the hyperupcall source.

It is important to note that a hyperupcall cannot invoke guest OS functions directly, since that code has not been secured by the framework. However, OS functions can be compiled into hyperupcalls and be integrated in the verified code.

3.2 Compilation

Once the hyperupcall has been written, it needs to be compiled into eBPF bytecode before the guest can register it with the hypervisor. Our framework generates this bytecode as part of the guest OS build process by running the hyperupcall C code through Clang and the eBPF LLVM backend, with some modifications to assist with address translation and verification:

Input field	Function
event	Event specific data including event ID.
hva	Host virtual address (HVA) in which the guest memory is mapped.
guest_mask	Guest address mask to mask bits which are higher than the guest memory address-width. Used for verification (§ 3.4).
vcpus	Pointers to the hypervisor VCPU data structure, if the event is associated with a certain VCPU, or a pointer to the guest OS data structure. Inaccessible to the hyperupcall, but used by helper functions.
vcpu_reg	Frequently accessed VCPU registers: instruction pointer and VCPU ID.
env	Environment variables, provided by the guest during hyperupcallregistration. Used to set address randomization offsets.

Table 4: *Hyperupcall context data.* These fields are populated by the hypervisor when a hyperupcall is called.

Guest memory access. To access guest memory, we use eBPF's direct packet access (DPA) feature, which was designed to allow programs to access network packets safely and efficiently without the use of helper functions. Instead of passing network packets, we utilize this feature by treating the guest as a "packet". Using DPA in this manner required a bug fix [2] to the eBPF LLVM backend, as it was written with the assumption that packet sizes are ≤64KB.

Address translations. Hyperupcalls allow the hypervisor to seamlessly use guest virtual addresses (GVAs), which makes it appear as if the hyperupcall was running in the guest. However, the code is actually executed by the hypervisor, where host virtual address (HVAs) are used, rendering guest pointers invalid. To allow the use of guest pointers transparently in the host context, these pointers therefore need to be translated from GVAs into HVAs. We use the compiler to make these translations.

To make this translation simple, the hypervisor maps the GVA range contiguously in the HVA space, so address translations can easily be done by adjusting the base address. As the guest might need the hyperupcall to access multiple contiguous GVA ranges—for example, one for the guest 1:1 direct mapping and of the OS text section [37]—our framework annotates each pointer with its respective "address space" attribute. We extend the LLVM compiler to use this information to inject eBPF code that converts each of the pointer from GVA to HVA by a simple subtraction operation. It should be noted that the generated code safety is not assumed by the hypervisor and is verified when the hyperupcall is registered.

Bound Checks. The verifier rejects code with direct memory accesses unless it can ensure the memory accesses are within the "packet" (in our case, guest memory) bounds. We cannot expect the hyperupcall programmer to perform the required checks, as the burden of adding them is substantial. We therefore enhance the compiler to automatically add code that performs bound checks prior to each memory access, allowing verification to pass. As we note in Section 3.4, the bounds checking is done using masking and not branches to ease verification.

Context caching. Our compiler extension introduces intrinsics to get a pointer to the context or to read its data. The context is frequently needed along the callback for calling helper functions and for translating GVAs. Delivering the context as a function parameter requires intrusive changes and can prevent sharing code between the guest and its hyperupcall. Instead, we use the compiler to cache the context pointer in one of the registers and retrieve it when needed.

3.3 Registration

After a hyperupcall is compiled into eBPF bytecode, it is ready to be registered. Guests can register hyperupcalls at any time, but most hyperupcalls are registered when the guest boots. The guest provides the hyperupcall event ID, hyperupcall bytecode and the virtual memory the hyperupcall will use. Each parameter is described below:

Hyperupcall event ID. ID of the event to handle.

Memory registration. The guest registers the virtual contiguous memory regions used by the hyperupcall. For global hyperupcalls, this memory is restricted to a maximum of 2% of the guest's total physical memory (configurable and enforced by the hypervisor).

Hyperupcall bytecode. The guest provides a pointer to the hyperupcall bytecode with its size.

3.4 Verification

The hypervisor verifies that each hyperupcall is safe to execute at registration time. Our verifier is based on the Linux eBPF verifier and checks three properties of the hyperupcall: memory accesses, number of runtime instructions, and helper functions used.

Ideally, verification is *sound*, ensuring only safe code passes verification, and *complete*, successfully verifying any safe program. While soundness cannot be compromised as it might jeopardize the system safety, many verification systems, including eBPF, sacrifice completeness to keep the verifier simple. In practice, the verifier requires programs to be written in a certain way to pass verification [66], and even then verification can fail due to path explosion. These limitations are at odds of our goal of making hyperupcalls simple to build.

We discuss the properties our verifier checks below, and how we simplify these checks to make verification as straightforward as possible.

Bounded runtime instructions. For global hyperupcalls, the eBPF verifier ensures that any possible execution of the hyperupcall contains a limited number of instructions, which is set by the hypervisor (defaulted to 4096). This ensures that the hypervisor can execute the hyperupcall in a timely manner, and that there are no infinite loops which can cause the hyperupcall not to exit.

Memory access verification. The verifier ensures that memory accesses only occur in the region bounded by the "packet", which in a hyperupcall is the virtual memory region provided during registration. As noted before, we enhance the compiler to automatically add code that proves to the verifier that each memory access is safe.

However, adding such code naively results in frequent verification failures. The current Linux eBPF verifier is quite limited in its ability to verify the safety of memory accesses, as it requires that they will be preceded by compare and branch instructions to prevent out of bound accesses. The verifier explores the possible execution paths and ensures their safety. Although the verifier employs various optimizations to prune branches and avoid walking every possible branch, verification often exhausts available resources and fails as we and others have experienced [65].

Therefore, instead of using compare and branch to ensure memory access safety, our enhanced compiler adds code that masks memory accesses offset within each range, preventing out-of-bounds memory accesses. We enhance the verifier to recognize this masking as safe. After applying this enhancement, all the programs we wrote passed verification.

Helper function safety. Hyperupcalls may call helper functions to both improve performance and to help limit the number of runtime instructions. Helper functions are a standard eBPF feature and the verifier enforces the helper functions which can be called, which may vary from event to event depending on hypervisor policy. For example, the hypervisor may disallow the use of `flush_tlb_vcpu` during memory reclamation, as it may block the system for an extended amount of time.

The verifier checks to ensure that the inputs to the helper function are safe, ensuring that the helper function only accesses memory which it is permitted to access. While these checks could be done in the helper function, new eBPF extensions allow the verifier to statically verify the helper function inputs. Furthermore, the hypervisor can also set a policy for inputs on a per-event basis (e.g, `memcpy` size for global hyperupcalls).

The number and complexity of helper functions should be limited as well, as they become part of the trusted computing base. We therefore only introduce simple helper functions, which mostly rely on code that the guest can already trigger today directly or indirectly, for example interrupt injection.

eBPF security. Two of the proof-of-concept exploits of the recently discovered “Spectre” hardware vulnerabilities [38, 30] targeted eBPF, which might raise concerns about eBPF and hyperupcall safety. While exploiting these vulnerabilities is simpler if an attacker can run unprivileged code in privileged context, just as hyperupcalls do, discovered attacks can be prevented [63]. In fact, these security vulnerabilities can make hyperupcalls more compelling as their mitigation techniques (e.g, return stack buffer stuffing [33]) induce extra overheads when context switches take place using traditional paravirtual mechanisms such as upcalls and hypercalls.

3.5 Execution

Verified hyperupcalls are installed into a per guest hyperupcall table. Once the hyperupcall has been registered and verified, the hypervisor executes hyperupcalls in response to events.

Hyperupcall patching. To avoid the overhead of testing whether hyperupcall is registered, the hypervisor uses a code patching technique, known in Linux as “static keys” [12]: a no-op instruction is set on each of the hypervisor hyperupcall invocation code only when hyperupcalls are registered.

Accessing remote VCPU state. Some hyperupcalls read or modify the state of remote VCPUs. These VCPUs may not be running or their state may be accessed by a different thread of the hypervisor. Even if the remote VCPU is preempted, the hypervisor may have already read some registers and not expect them to change until the VCPU resumes execution. If the hyperupcall writes to remote VCPU registers, it may break the hypervisor invariants and even introduce security issues.

Furthermore, reading remote VCPU registers can induce high overheads, as part of the VCPU state may be cached in another CPU, and must be written back to memory first if the VCPU state is to be read. More importantly, in Intel CPUs the VCPU state cannot be accessed by common instructions, and the VCPU must be “loaded” first before its state can be accessed by using special instructions (`VMREAD` and `VMWRITE`). Switching the loaded VCPU incurs significant overhead, which roughly 1800 cycles on our system.

For performance, we define synchronization points where the hypervisor is commonly preempted, and accessing the VCPU state is known to be safe. At these points we “decache” VCPU registers from the VMCS and write them to the memory so the hyperupcall can read them. The hyperupcall writes to remote VCPU registers and updates the decached value to flag the hypervisor to reload the register values into the VMCS before resuming that VCPU. Hyperupcalls that access remote VCPUs are executed on a best-effort basis, running only if the VCPU is in a synchronization point. The remote VCPU is prevented from resuming execution while the hyperupcall is running.

Using guest OS locks. Some of the OS data-structures are protected by locks. Hyperupcalls that require consistent guest OS data structure view should abide the synchronization scheme that the guest OS dictates. Hyperupcall, however, can only acquire locks opportunistically, since a VCPU might be preempted while holding a lock. The lock implementation might need to be adapted to support locking by an external entity, different than any VCPU. Releasing a lock can require relatively large code to handle slow-paths, which might prevent the timely verification of the hyperupcall.

While various ad-hoc solutions may be proposed, it seems a complete solution requires the guest OS locks to be hyperupcall-aware. It also necessitates support for calling eBPF function from eBPF code to avoid inflated code size that might cause verification failures. Since this support has been added very recently, our implementation does not include lock support.

4 Use Cases and Evaluation

Our evaluation is guided by the following questions:

- What are the overheads of using verified code (eBPF) versus native code? (§4.1).
- How do hyperupcalls compare to other paravirtual mechanisms (§4.3, 4.2, 4.5)?
- How can hyperupcalls enhance not only the performance (§4.3, 4.2) but also the security (§4.5) and debuggability (§4.4) of virtualized environments?

Testbed. Our testbed consists of a 48 core dual-socket Dell PowerEdge R630 server with Intel E5-2670 CPUs, a Seagate ST1200 disk, which runs Ubuntu 17.04 with Linux kernel v4.8. The benchmarks are run on guests with 16 VCPUs and 8GB of RAM. Each measurement was performed 5 times and the average result is reported.

Hyperupcall prototype. We implemented a prototype for hyperupcall support on Linux v4.8 and KVM, the hypervisor which is integrated in Linux. Hyperupcalls are compiled through a patched LLVM 4, and are verified through the Linux kernel eBPF verifier with the patches we described in §3. We enable the Linux eBPF “JIT” engine , which compiles the eBPF code to native machine code after verification. The correctness of the BPF JIT

	h-visor	runtime (cycles)		eBPF	C
use case	event	h-upcall	native	instr.	SLoC
discard § 4.3	reclaim	185	147	357	32
tracing § 4.4	exit	568	336	3308	889
TLB § 4.2	interrupt	395	530	111	112
protect § 4.5	exit	43	25	119	74
	map	108	92	170	52

Table 5: Evaluated hyperupcall use cases, comparison of runtime, eBPF instructions and number of lines of code.

engine has been studied and can be verified [74].

Use cases. We evaluate four hyperupcall use cases as listed in Table 5. Each use case demonstrates the use of hyperupcalls on different hypervisor events, and uses hyperupcalls of varying complexity.

4.1 Hyperupcall overheads

We evaluate the overheads of using verified code to service hypervisor requests by comparing the runtime of a hyperupcall versus native code with the same function (Table 5). Overall, we find that the absolute overhead of the verified code relative to native is small (< 250 cycles). For the TLB use case which handles TLB shootdown to inactive cores, our hyperupcall runs faster than native code since the TLB flush is deferred. The overhead of verifying a hyperupcall is minimal. For the longest hyperupcall (tracing), verification took 67ms.

4.2 TLB Shootdown

While interrupt delivery to VCPUs can usually be done efficiently, there is a significant penalty if the target VCPU is not running. This can occur if CPUs are overcommitted and scheduling the target VCPU requires preempting another VCPU. With synchronous interprocessor interrupts (IPIs), the sender resumes execution only after the receiver indicates the IPI was delivered and handled, resulting in prohibitive overheads.

The overhead of IPI delivery is most notable in the case of translation lookaside buffer (TLB) shootdowns, a software protocol that OSs use to keep TLBs—caches of virtual to physical address mapping—coherent. As common CPU architectures (e.g., x86) do not keep TLBs coherent in hardware, an OS thread that modifies a mapping sends an IPI to other CPUs that may cache the mapping, and these CPUs then flush their TLBs.

We use hyperupcalls to handle this scenario by registering a hyperupcall which handles TLB shootdowns when interrupts are delivered to a VCPU. The hypervisor provides that hyperupcall with the interrupt vector and the target VCPU after ensuring it is in quiescent state. Our hyperupcall checks whether this vector is the "remote function invocation" vector and whether the function pointer equals to the OS TLB flush function. If it does, it runs this function with few minor modifications: (1) instead of flushing the TLB using native instruction, the TLB flush is performed using a helper function, which defers it to the next VCPU re-entry; (2) TLB flush is performed even when the VCPU interrupts are disabled, as experimentally it improves performance.

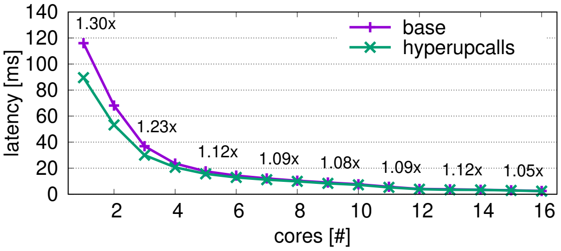


Figure 2: The latency of Apache when CPUs are overcommitted, with and without a hyperupcall that handle interrupts to preempted VCPUs. Numbers above data points indicate speedup over base.

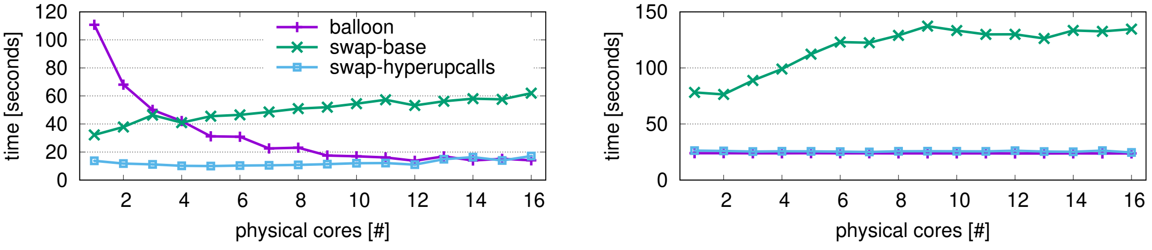
Admittedly, an alternative solution is available: introducing a hypercall that delegates TLB flushes to the hypervisor [52]. Although this solution can prevent TLB flushes, it requires a different code path, which may introduce hidden bugs [43], complicate the integration with OS code or introduce additional overheads [44]. This solution is also limited to TLB flushes, and cannot deal with other interrupts, for example, rescheduling IPIs.

Evaluation We run Apache Web server [23] in a guest using the default `mpm_event` module, which runs multithreaded workers to handle incoming requests. To measure performance, we use ApacheBench, an Apache HTTP server benchmarking tool, generating 10k requests using 16 connections, and measuring the request latency. The results, which are shown in Figure 2, show hyperupcalls reduce the latency by up to 1.3×. It might appear surprising that performance improves even when the physical CPUs are not oversubscribed. However, as VCPUs are often momentarily idle in this benchmark, they can also trigger an exit to the hypervisor.

4.3 Discarding Free Memory

Free memory, by definition, holds no needed data and can be discarded. If the hypervisor knows what memory is free in the guest, it can discard it during memory reclamation, snapshotting, live migration or lock-step execution [20] and avoid I/O operations for saving and restoring their content. Information on which memory pages are free, however, is held by the guest and unavailable to the hypervisor due to the semantic gap.

Throughout the years several mechanisms have been proposed to inform the hypervisor which memory pages



(a) reclaim

(b) refault

Figure 3: Time of guest memory reclaim of 7GB and refault, when reading a 4GB file, when CPUs are overcommitted. The x-axis shows the number of physical cores available. (a) As the number of physical cores decrease (and overcommitment increases), the time to reclaim memory increases. (b) refaulting free memory incurs a significant penalty for uncooperative swapping (swap-base) on its own because it swaps out active and free pages.

are free using paravirtualization. These solutions, however, either couple the guest and hypervisor [60]; induce overheads due to frequent hypercalls [41] or are limited to live migration [73]. All of these mechanisms suffer for an inherent limitation: without coupling the guest and the hypervisor, the guest needs to communicate to the hypervisor which pages are free.

In contrast, a hypervisor that supports hyperupcalls does not need to be notified about free pages. Instead, the guest sets a hyperupcall that describes whether a page is discardable based on the page metadata (Linux's `struct page`) and is based in Linux on the `is_free_buddy_page` function. When the hypervisor performs an operation that can benefit from discarding a free guest memory page such as reclaiming a page, the hypervisor invokes this hyperupcall to check whether the page is discardable. The hyperupcall is also called when the page is already unmapped, preventing a race in which it is discarded when it is no longer free.

Checking whether a page can be discarded must be done through a global hyperupcall, since the answer must be provided in a bounded and short time. As a result, the guest can only register part of its memory to be used by the hyperupcall, since this memory is never paged out to ensure timely execution of the hyperupcall. Our Linux guest registers the memory of the pages' metadata, which accounts to about 1.6% of the guest's physical memory.

Evaluation. To evaluate the performance of the "memory discard" hyperupcall, we measure its impact on a guest whose memory is reclaimed due to memory pressure. When memory is scarce, hypervisors can perform "uncooperative swapping"—directly reclaim guest memory and swap it out to disk. This approach, however, often leads to suboptimal reclamation decisions. Alternatively, hypervisors can use *memory ballooning*, a paravirtual mechanism in which a guest module is informed on host memory pressures and causes the guest to reclaim memory directly [71]. The guest can then make knowledgeable reclamation decisions and discard free pages. Although memory ballooning usually performs well, performance suffers when memory needs to be abruptly reclaimed [4, 6] or when the guest disk is set on a network attached storage [68], and it is therefore not used under high memory pressure [21].

To evaluate memory ballooning, uncooperative swapping and swapping with hyperupcalls we run a scenario in which memory and physical CPU need to be abruptly reclaimed, such as to accommodate a new guest. In the guest, we start and exit "memhog", making 4GB available to be reclaimed in the guest. Next, we make the guest busy by running a CPU intensive task with low memory footprint - the SysBench CPU benchmark, which computes primes using all VCPUs [39].

Now, with the the system busy, we simulate the need to reclaim resources to start a new guest by increasing memory and CPU overcommitment. We lower the number of physical CPUs available to the guest and restrict it to only 1GB of memory. We measure the time it takes to reclaim memory against the number of physical CPUs that were allocated for the guest (Figure 3a). This simulates a new guest starting up. Then, we stop increasing memory pressure and measure the time to run a guest application with a large memory footprint using the SysBench file read benchmark on 4GB (Figure 3b). This simulates the guest reusing pages that have been reclaimed by the hypervisor.

Ballooning reclaims memory slowly (up to 110 seconds) when physical CPUs are overcommitted, as the memory reclamation operations compete with the CPU intensive tasks on CPU time. Uncooperative swapping (swap-base) can reclaim faster (32 seconds), but as it is oblivious to whether memory pages are free, it incurs higher overhead in refaulting guest free pages. In contrast, when hyperupcalls are used, the hypervisor can promote free pages' reclamation and discard them, thereby reclaiming memory up to 8 times faster than balloon, with only 10% slowdown in refaulting the memory.

CPU overcommitment, of course, is not the only scenario where ballooning is non-responsive or unusable. Hypervisors refrain from ballooning when memory pressure is very high, and use host-level swapping instead [67]. It is possible for hyperupcalls to operate synergistically with ballooning: the hypervisor may use the balloon normally and use hyperupcalls when resource pressures are high or the balloon is not responding.

4.4 Tracing

Event tracing is an important tool for debugging correctness and performance issues. However, collecting traces for virtualized workloads is somewhat limited. Traces collected inside a guest do not show hypervisor events, such as when a VM-exit is forced, which can have significant effect on performance. For traces that are collected in the hypervisor to be informative, they require knowledge about guest OS symbols [15]. Such traces cannot be collected in cloud environments. In addition, each trace collects only part of the events and does not show how guest and hypervisor events interleave.

To address this issue, we run the Linux kernel tracing tool, `ftrace` [57], inside a hyperupcall. `Ftrace` is well suited to run in a hyperupcall. It is simple, lockless, and built to enable concurrent tracing in multiple contexts: non-maskable interrupt (NMI), hard and soft interrupt handlers and user processes. As a result, it was easily be adapted to trace hypervisor events concurrently with guest events. Using the `ftrace` hyperupcall, the guest can trace both hypervisor and guest events in one unified log, easing debugging. Since tracing all events use only guest logic, new OS versions can change the tracing logic, without requiring hypervisor changes.

Evaluation. Tracing is efficient, despite the hyperupcallcomplexity (3308 eBPF instructions), as most of the code deals with infrequent events that handles situations in which trace pages fill up. Tracing using hyperupcalls is slower than using native code by 232 cycles, which is still considerably shorter time than the time a context switch between the hypervisor and the guest takes.

Tracing is a useful tool for performance debugging, which can expose various overheads [79]. For example, by registering the `ftrace` on the VM-exit event, we see that many processes, including short-lived ones, trigger multiple VM exits due to the execution of the `CPUID` instruction, which enumerates the CPU features and must be emulated by the hypervisor. We find that the GNU C Library, which is used by most Linux applications, uses `CPUID` to determine the supported CPU features. This overhead could be prevented by extending Linux virtual dynamic shared object (vDSO) for applications to query the supported CPU features without triggering an exit.

4.5 Kernel Self-Protection

One common security hardening mechanisms that OSs employ is “self-protection”: OS code and immutable data write protection. However, this protection is done using page tables, allowing malware to circumvent it by modifying page table entries. To prevent such attacks, the use of nested page tables has been suggested, as these tables are inaccessible from the guest [50].

However, nesting can only provide a limited number of policies and for example, cannot whitelist guest code that is allowed to access protected memory. Hyperupcalls are much more expressive, allowing the guest to specify memory protection in a flexible manner.

We use hyperupcalls to provide hypervisor-level guest kernel self-protection, which can be easily modified to accommodate complex policies. In our implementation the guest sets a bitmap which marks protected pages, and registers hyperupcall on exit events, which checks the exit reason, whether a memory access occurred and if the guest attempted to write to protected memory according to the bitmap. If there is an attempt to access protected memory, a VM shutdown is triggered. The guest sets an additional hyperupcall on the “page map” event, which queries the required protection of the guest page frames. This hyperupcall prevents situations in which the hypervisor proactively prefaults guest memory.

Evaluation. This hyperupcall code is simple, yet incurs overhead of 43 cycles per exit. Arguably, only workloads which already experience very high number of context switches would be affected by the additional overheads. Modern CPUs prevent such frequent switches.

5 Conclusion

Bridging the semantic gap is critical performance and for the hypervisor to provide advanced services to guests. Hypercalls and upcalls are now used to bridge the gap, but they have several drawbacks: hypercalls cannot be initiated by the hypervisor, upcalls do not have a bounded runtime, and both incur the penalty of context switches. Introspection, an alternative which avoids context switches can be unreliable as it relies on observations instead of an explicit interface. Hyperupcalls overcome these limitations by allowing the guest to expose its *logic* to the hypervisor, avoiding a context switch by enabling the hyperupcall to safely execute guest logic directly.

We have built a complete infrastructure for developing hyperupcalls which allow developers to easily add new paravirtual features using the codebase of the OS. We have written and evaluated several hyperupcalls and show hyperupcalls improve virtualized performance by up to 2×, ease debugging of virtualized workloads and improve VM security.

References

[1] Ferrol Aderholdt, Fang Han, Stephen L Scott, and Thomas Naughton. Efficient checkpointing of virtual machines using virtual machine introspection. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 414–423, 2014.

[2] Nadav Amit. Patch: Wrong size-extension check in BPFDAGToDAGISel::SelectAddr. `https://lists.iovisor.org/pipermail/iovisor-dev/2017-April/000723.html`, 2017.

[3] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, 2011.

[4] Nadav Amit, Dan Tsafrir, and Assaf Schuster. VSwapper: A memory swapper for virtualized environments. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 349–366, 2014.

[5] Nadav Amit, Michael Wei, and Cheng-Chun Tu. Hypercallbacks: Decoupling policy decisions and execution. In *ACM Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 37–41, 2017.

[6] Kapil Arya, Yury Baskakov, and Alex Garthwaite. Tesseract: reconciling guest I/O and hypervisor swapping in a VM. In *ACM SIGPLAN Notices*, volume 49, pages 15–28, 2014.

[7] Anish Babu, MJ Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. System performance evaluation of para virtualization, container virtualization, and full virtualization using Xen, OpenVX, and Xenserver. In *IEEE International Conference on Advances in Computing and Communications (ICACC)*, pages 247–250, 2014.

[8] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *IEEE Symposium on Reliable Distributed Systems*, pages 82–91, 2010.

[9] Mirza Basim Baig, Connor Fitzsimons, Suryanarayanan Balasubramanian, Radu Sion, and Donald E. Porter. Cloudflow: Cloud-wide policy enforcement using fast vm introspection. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–164, 2014.

[10] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(5):670–684, 2011.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[12] Jason Baron. Static keys. Linux-4.8:Documentation/static-keys.txt, 2012.

[13] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the spin operating system. *ACM SIGOPS Operating Systems Review (OSR)*, 29(5):267–283, 1995.

[14] Big Switch Networks. Userspace eBPF VM. `https://github.com/iovisor/ubpf`, 2015.

[15] Martim Carbone, Alok Kataria, Radu Rugina, and Vivek Thampi. VProbes: Deep observability into the ESXi hypervisor. VMware Technical Journal `https://labs.vmware.com/vmtj/vprobes-deep-observability-into-the-esxi-hypervisor`, 2014.

[16] Andrew Case, Lodovico Marziale, and Golden G Richard. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7:S32–S40, 2010.

[17] Peter M Chen and Brian D Noble. When virtual is better than real. In *IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 133–138, 2001.

[18] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. Introspection-based memory de-duplication and migration. In *ACM SIGPLAN Notices*, volume 48, pages 51–62, 2013.

[19] Tzi-cker Chiueh, Matthew Conover, and Bruce Montague. Surreptitious deployment and execution of kernel agents in windows guests. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 507–514, 2012.

[20] Eddie Dong and Will Auld. COLO: COarse-grain LOckstepping virtual machine for non-stop service. Linux Plumbers Conference, 2012.

[21] Craig Thomas Ellrod. *Optimizing Citrix XenDesktop for High Performance*. Packt Publishing, 2015.

[22] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. *Exokernel: An operating system architecture for application-level resource management*, volume 29. 1995.

[23] Roy T. Fielding and Gail Kaiser. The Apache HTTP server project. *IEEE Internet Computing*, 1(4):88–90, 1997.

[24] Yangchun Fu and Zhiqiang Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *IEEE Symposium on Security and Privacy (SP)*, pages 586–600, 2012.

[25] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *The Network and Distributed System Security Symposium (NDSS)*, volume 3, pages 191–206, 2003.

[26] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process implanting: A new active introspection framework for virtualization. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 147–156, 2011.

[27] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. Efficient and scalable paravirtual I/O system. *USENIX Annual Technical Conference (ATC)*, 2013.

[28] Jennia Hizver and Tzi-cker Chiueh. Real-time deep virtual machine introspection and its applications. In *ACM SIGPLAN Notices*, volume 49, pages 3–14, 2014.

[29] Owen S Hofmann, Alan M Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring operating system kernel integrity with OSck. In *ACM SIGARCH Computer Architecture News (CAN)*, volume 39, pages 279–290, 2011.

[30] Jann Horn. Speculative execution, variant 4: speculative store bypass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`, 2018.

[31] Nur Hussein. Randomizing structure layout. `https://lwn.net/Articles/722293/`, 2017.

[32] Amani S Ibrahim, James Hamlyn-Harris, John Grundy, and Mohamed Almorsy. Cloudsec: a security monitoring appliance for virtual machines in the iaas cloud model. In *IEEE International Conference on Network and System Security (NSS)*, pages 113–120, 2011.

[33] Intel. Retpoline, a branch target injection mitigation. `https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf?source=techstories.org`, 2018.

[34] IO Visor Project. BCC - tools for BPF-based Linux IO analysis, networking, monitoring, and more. `https://github.com/iovisor/bcc`, 2015.

[35] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In *ACM Conference on Computer and Communications Security (CCS)*, pages 128–138, 2007.

[36] Stephen T Jones, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference (ATC)*, pages 1–14, 2006.

[37] Andi Kleen. Linux virtual memory map. Linux-4.8:Documentation/x86/x86_64/mm.txt, 2004.

[38] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[39] Alexey Kopytov. SysBench: a system performance benchmark. `https://github.com/akopytov/sysbench`, 2017.

[40] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA)*, 2007.

[41] Nitesh Narayan Lal. KVM guest page hinting RFC. KVM mailing list `http://www.spinics.net/lists/kvm/msg153666.html`, 2017.

[42] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. *Pre-virtualization: Slashing the cost of virtualization*. Universität Karlsruhe, Fakultät für Informatik, 2005.

[43] Andrew Lutomirski. Fix flush_tlb_page() on Xen. Linux Kernel Mailing List, `https://patchwork.kernel.org/patch/9693379/`, 2017.

[44] Andrew Lutomirski. Patch: x86/hyper-v: use hypercall for remote TLB flush. Linux Kernel Mailing List, `http://www.mail-archive.com/linux-`

kernel@vger.kernel.org/msg1402180.html, 2017.

[45] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, volume 48, pages 461–472, 2013.

[46] Microsoft. Hypervisor top level functional specification v5.0b, 2017.

[47] Aleksandar Milenkoski, Bryan D Payne, Nuno Antunes, Marco Vieira, and Samuel Kounev. Experience report: an analysis of hypercall handler vulnerabilities. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–111, 2014.

[48] Preeti Mishra, Emmanuel S Pilli, Vijay Varadharajan, and Udaya Tupakula. Intrusion detection techniques in cloud environment: A survey. *Journal of Network and Computer Applications*, 77:18–47, 2017.

[49] Quentin Monnet. Rust virtual machine and JIT compiler for eBPF programs. https://github.com/qmonnet/rbpf, 2017.

[50] Jun Nakajima and Sainath Grandhi. Kernel protection using hardware based virtualization. KVM Forum, 2016.

[51] George C Necula. Proof-carrying code. In *ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL)*, pages 106–119, 1997.

[52] Jiannan Ouyang, John R Lange, and Haoqiang Zheng. Shoot4U: Using VMM assists to optimize TLB operations on preempted vCPUs. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, 2016.

[53] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy (SP)*, pages 233–247, 2008.

[54] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library OS from the top down. In *ACM SIGPLAN Notices*, volume 46, pages 291–304, 2011.

[55] Adit Ranadive, Ada Gavrilovska, and Karsten Schwan. Ibmon: monitoring VMM-bypass capable infiniband devices using memory introspection. In *Workshop on System-level Virtualization for HPC (HPCVirt)*, pages 25–32, 2009.

[56] Dannies M. Ritchie and Ken Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1929, 1978.

[57] Steven Rostedt. Debugging the kernel using Ftrace. LWN.net, http://lwn.net/Articles/365835/, 2009.

[58] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review (OSR)*, 42(5):95–103, 2008.

[59] Joanna Rutkowska and Alexander Tereshkin. Bluepilling the Xen hypervisor. *BlackHat USA*, 2008.

[60] Martin Schwidefsky, Hubertus Franke, Ray Mansell, Himanshu Raj, Damian Osisek, and JongHyuk Choi. Collaborative memory management in hosted Linux environments. In *Ottawa Linux Symposium (OLS)*, volume 2, pages 313–330, 2006.

[61] Jiangyong Shi, Yuexiang Yang, and Chuan Tang. Hardware assisted hypervisor introspection. *SpringerPlus*, 5(1):647, 2016.

[62] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-NFV: Securing NFV states by using SGX. In *ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV)*, pages 45–48, 2016.

[63] Alexei Starovoitov. Patch: BPF: introduce BPF_JIT_ALWAYS_ON config. Linux Kernel Mailing List, https://lkml.org/lkml/2018/1/9/858, 2018.

[64] Sahil Suneja, Canturk Isci, Vasanth Bala, Eyal De Lara, and Todd Mummert. Non-intrusive, out-of-band and out-of-the-box systems monitoring in the cloud. In *ACM SIGMETRICS Performance Evaluation Review (PER)*, volume 42, pages 249–261, 2014.

[65] William Tu. bpf invalid stack off=-528. IO Visor mailing list https://lists.iovisor.org/pipermail/iovisor-dev/2017-April/000724.html, 2017.

[66] William Tu. Direct packet access boundary check. IO-Visor mailing list https://lists.iovisor.org/pipermail/iovisor-dev/2017-January/000604.html, 2017.

[67] vSphere memory states. `http://www.running-system.com/vsphere-6-esxi-memory-states-and-reclamation-techniques/`.

[68] Best practices for running VMware vSphere on network attached storage. `https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-nfs-bestpractices-white-paper-en.pdf`, 2009.

[69] VMware. open-vm-tools. `https://github.com/vmware/open-vm-tools`, 2017.

[70] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review (OSR)*, volume 27, pages 203–216, 1994.

[71] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review (OSR)*, 36(SI):181–194, 2002.

[72] Gary Wang, Zachary John Estrada, Cuong Manh Pham, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. Hypervisor introspection: A technique for evading passive virtual machine monitoring. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[73] Wei Wang. Support reporting free page blocks patch. Linux Kernel Mailing List, `https://lkml.org/lkml/2017/7/12/253`, 2017.

[74] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 33–47, 2014.

[75] Rafal Wojtczuk. Analysis of the attack surface of Windows 10 virtualization-based security. Black-Hat USA, 2016.

[76] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, and Mazin S Yousif. Black-box and gray-box strategies for virtual machine migration. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 7, pages 17–17, 2007.

[77] Britta Wuelfing. Kroah-Hartman: Remove Hyper-V driver from kernel? Linux Magazine `http://www.linux-magazine.com/Online/News/Kroah-Hartman-Remove-Hyper-V-Driver-from-Kernel`, 2009.

[78] Xen Project. XenParavirtOps. `https://wiki.xenproject.org/wiki/XenParavirtOps`, 2016.

[79] Haoqiang Zheng and Jason Nieh. WARP: Enabling fast CPU scheduler development and evaluation. In *IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS)*, pages 101–112, 2009.

AIQL: Enabling Efficient Attack Investigation from System Monitoring Data

Peng Gao[1] Xusheng Xiao[2] Zhichun Li[3] Kangkook Jee[3] Fengyuan Xu[4]
Sanjeev R. Kulkarni[1] Prateek Mittal[1]
[1]*Princeton University* [2]*Case Western Reserve University* [3]*NEC Laboratories America, Inc.*
[4]*National Key Lab for Novel Software Technology, Nanjing University*
[1]*{pgao,kulkarni,pmittal}@princeton.edu* [2]*xusheng.xiao@case.edu* [3]*{zhichun,kjee}@nec-labs.com* [4]*fengyuan.xu@nju.edu.cn*

Abstract

The need for countering Advanced Persistent Threat (APT) attacks has led to the solutions that ubiquitously monitor system activities in each host, and perform timely attack investigation over the monitoring data for analyzing attack provenance. However, existing query systems based on relational databases and graph databases lack language constructs to express key properties of major attack behaviors, and often execute queries inefficiently since their semantics-agnostic design cannot exploit the properties of system monitoring data to speed up query execution.

To address this problem, we propose a novel query system built on top of existing monitoring tools and databases, which is designed with novel types of optimizations to support timely attack investigation. Our system provides (1) domain-specific *data model and storage* for scaling the storage, (2) a domain-specific query language, *Attack Investigation Query Language (*AIQL*)* that integrates critical primitives for attack investigation, and (3) an optimized query engine based on the characteristics of the data and the semantics of the queries to efficiently schedule the query execution. We deployed our system in NEC Labs America comprising 150 hosts and evaluated it using 857 GB of real system monitoring data (containing 2.5 billion events). Our evaluations on a real-world APT attack and a broad set of attack behaviors show that our system surpasses existing systems in both efficiency (124x over PostgreSQL, 157x over Neo4j, and 16x over Greenplum) and conciseness (SQL, Neo4j Cypher, and Splunk SPL contain at least 2.4x more constraints than AIQL).

1 Introduction

Advanced Persistent Threat (APT) attacks are sophisticated (involving many individual attack steps across many hosts and exploiting various vulnerabilities) and stealthy (each individual step is not suspicious enough), plaguing many well-protected businesses [9, 11, 15, 18, 27, 30]. A recent massive Equifax data breach [11] has exposed the sensitive personal information of 143 million US customers. In order for enterprises to counter advanced attacks, recent approaches based on *ubiquitous system monitoring* have emerged as an important solution for monitoring system activities and performing attack investigation [37,42,47–49,54,57,58]. System monitoring observes *system calls* at the kernel level to collect system-level events about system activities. Collection of system monitoring data enables security analysts to investigate these attacks by *querying risky system behaviors* over the historical data [71].

Although attack investigation is performed after the attacks compromise enterprises' security, it is a considerably time-sensitive task due to two major reasons. First, advanced attacks include a sequence of steps and are performed in multiple stages. A timely attack investigation can help understand all attack behaviors and prevent the further damage of the attacks. Second, understanding the attack sequence is crucial to correctly patch the systems. A timely attack investigation can pinpoint the vulnerable components of the systems and protect the enterprises from future attacks of the same types.

Challenges: However, there are two major challenges for building a query system to support security analysts in efficient and timely attack investigation.

Attack Behavior Specification: The system needs to provide a query language with specialized constructs for expressing various types of attack behaviors using system monitoring data: (1) **Multi-Step Attacks**: risky behaviors in advanced attacks typically involve activities that are related to each other based on either specific attributes (e.g., the same process reads a sensitive file and accesses the network) or temporal relationships (e.g., file read happens before network access), which requires language constructs to easily specify *relationships among activities*. In Fig. 1, the attacker runs `osql`

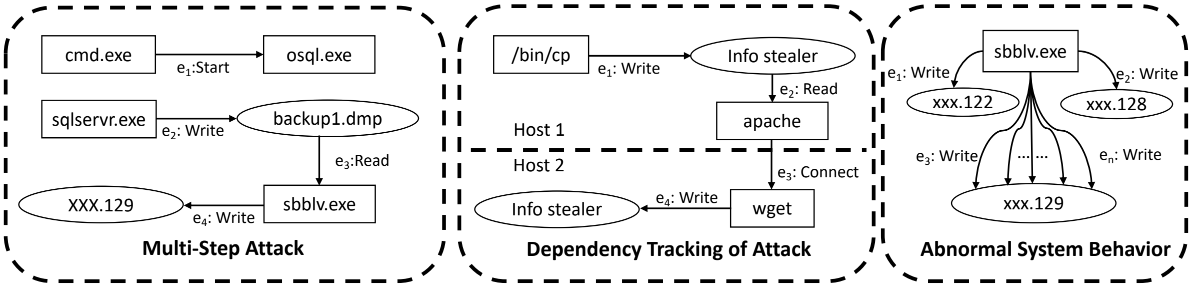


Figure 1: Major types of attack behaviors (events $e_1,\ldots,e_n$ are shown in ascending temporal order)

`.exe` to cause the database `sqlservr.exe` to dump its data into a file `backup1.dmp`. Later (i.e., `e3` happens after `e2`; temporal relationship), a malicious script `sbblv.exe` reads from the dump `backup1.dmp` (i.e., the same dump file in `e2` and `e3`; attribute relationship) and sends the data back to the attacker. (2) **Dependency Tracking of Attacks**: dependency analysis is often applied to track causality of data for discovering the "attack entry" (i.e., provenance) [48,49,61], which requires language constructs to *chain constraints among activities*. In Fig. 1, a malicious script `info_strealer` in Host 1 infects Host 2 via *network communications between* `apache` *and* `wget`. (3) **Abnormal System Behaviors**: frequency-based behavioral models are often required to express abnormal system behaviors, such as network access spikes [20, 29]. Investigating such spikes requires the system to support *sliding windows and statistical aggregation* of system activities, and compare the aggregate results with either *fixed thresholds (in absolute sense)* or the *historical results (in relative sense)*. In Fig. 1, a malicious script `sbblv.exe` sends a *large* amount of data to a particular destination `xxx.129`.[1]

Big-Data Security Analysis: System monitoring produces a huge amount of daily logs [55,69] (∼ 50 GB per day for 100 hosts), and the investigation of these attacks typically requires enterprises to keep at least a 0.5 ∼ 1 year worth of data [32]. Such *a big amount of security data* poses challenges for the system to meet the requirements of *timely* attack investigation.

Limitations of Existing Systems: Unfortunately, existing query systems do not address both of these inherent challenges in attack investigation. First, existing query languages in relational databases based on SQL and SPARQL [19,22,25] lack constructs for easily chaining constraints among relations. Graph databases such as Neo4j [16] and NoSQL tools such as MongoDB [38], Splunk [23], and ElasticSearch [10] are ineffective in expressing event relationships where two events have no common entities (e.g., e_1 and e_2 in Fig. 1). More importantly, none of these languages provide language constructs to express behavioral models with historical results. Second, system monitoring data is generated with a timestamp on a specific host in the enterprise, exhibiting strong *spatial and temporal properties*. However, none of these systems provide optimizations that exploit the domain specific characteristics of the data, missing opportunities to optimize the system for supporting timely attack investigation and often causing queries to run for hours (e.g., performance evaluation results in Sec. 6.2.2).

Contributions: We design and build a novel system for efficient attack investigation from system monitoring data. We build our system (∼ 50,000 lines of Java code) on top of existing system-level monitoring tools (i.e., auditd [28] and ETW [13]) for data collection and relational databases (i.e., PostgreSQL [19] and Greenplum [14]) for data storage and query. This enables our system to leverage the services provided by these mature infrastructures, such as data management, indexing mechanisms, recovery, and security. In particular, our system is designed with three novel types of optimizations. First, our system provides a domain-specific query language, *Attack Investigation Query Language (*AIQL*)*, which is optimized to express the three aforementioned types of attack behaviors. Second, our system provides domain-specific *data model and storage* for scaling the storage. Third, our system optimizes the query engine based on the characteristics of the monitoring data and the semantics of the queries to efficiently schedule the query execution. To the best of our knowledge, we are *the first to accelerate attack investigation via optimizing storage and query of system monitoring data.*

```
agentid = 1 // host id; spatial constraints
(at "01/01/2017") // temporal constraints
proc p1 start proc p2["%telnet%"] as evt1
proc p3 start ip ipp[dstport = 4444] as evt2
proc p4["%apache%"] read file f1["/var/www%"] as evt3
with p2 = p3,  // attribute relationship
evt1 before evt2, evt3 after evt2 // temporal
     relationships
return p1, p2, p4, f1
```

Query 1: AIQL Query for CVE-2010-2075 [5]

Domain-Specific Query Language (Sec. 4): Our AIQL language is designed for specifying the attack behaviors shown in Fig. 1 (i.e., Query 7 in Sec. 6.2.1, Query 3 in Sec. 4.2, and Query 5 in Sec. 6.2.1, respectively). Specifically, AIQL provides language constructs to specify re-

[1] While existing complex event processing systems [3, 12, 21] support similar features, they operate over stream rather than historical data stored in databases.

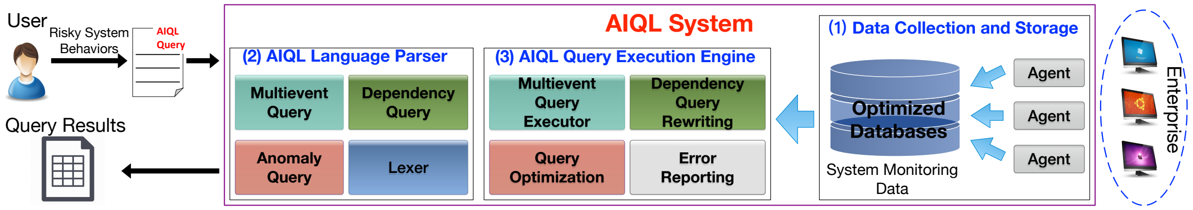


Figure 2: The AIQL system architecture

lationships among system activities (Sec. 4.1), chain constraints among activities (Sec. 4.2), and compute aggregate results in sliding time windows (Sec. 4.3). AIQL adopts the {*subject-operation-object*} syntax to represent system behavior patterns as events (e.g., `proc p1 write file f1`) and supports *attribute relationships* and *temporal relationships* of multiple events, as well as syntax shortcuts based on context-aware inference (Sec. 4.1). As shown in Query 1, AIQL can relate multiple system activities using spatial/temporal constraints and attribute/temporal relationships.

Data Model and Storage (Sec. 3.2): Our system models system monitoring data as a sequence of events, where each event describes how a process interacts with a system resource, such as writing to a file. More importantly, our system clearly identifies the spatial and temporal properties of the events, and leverages these properties to partition the data storage in both *spatial and temporal dimensions*. Such partitioning presents opportunities for parallel processing of query execution (Sec. 5.2).

Query Scheduling (Sec. 5): Our system identifies both *spatial* and *temporal* constraints in AIQL queries, and optimizes the query execution in two aspects: (1) for AIQL queries that involve multiple event patterns, our system prioritizes the search of event patterns with high pruning power, maximizing the reduction of irrelevant events as early as possible; (2) our system breaks down the query into independent sub-queries along temporal and spatial dimensions and executes them in parallel.

Evaluation: We deployed the AIQL system in NEC Labs America comprising 150 hosts. We performed a broad set of attack behaviors in the deployed environment, and evaluated the query performance and conciseness of AIQL against existing systems using 857 GB of real system monitoring data (16 days; 2.5 billion events): (1) our end-to-end efficiency evaluations on an APT attack case study (27 queries) show that AIQL surpasses both PostgreSQL (124x) and Neo4j (157x); (2) our performance evaluations show that the query scheduling employed by AIQL is efficient in both single-node databases (40x over PostgreSQL scheduling) and parallel databases (16x over Greenplum scheduling); (3) our conciseness evaluations on four major types of attack behaviors (19 queries) show that SQL, Neo4j Cypher, and Splunk SPL contain at least 2.4x more constraints, 3.1x more words, and 4.7x more characters than AIQL. All queries and a demo video are available on our *project website [1]*.

2 System Overview and Threat Model

Fig. 2 shows the AIQL system architecture: (1) we deploy monitoring agents across servers, desktops and laptops in the enterprise to monitor system activities by collecting information about system calls from kernels. The collected system monitoring data is then sent to the central server and stored in our optimized data storage (Sec. 3); (2) the language parser, implemented using ANTLR 4 [2], analyzes input queries and generates query contexts. A query context is an object abstraction of the input query that contains all the required information for the query execution. Multievent syntax, dependency syntax, and anomaly syntax are supported (Sec. 4); (3) the query execution engine executes the generated query contexts to search for the desired attack behaviors. Based on the data storage and the query semantics, domain-specific optimizations, such as relationship-based scheduling and temporal & spatial parallelization, are adopted to speedup the query execution (Sec. 5).

Threat Model: Our thread model follows the threat model of previous work [34, 48, 49, 54, 55]. We assume that kernel is trusted, and the system monitoring data collected from kernel is not tampered with [13, 28]. Any kernel-level attack that deliberately compromises security auditing systems is beyond the scope of this work.

3 Data Model and Storage

3.1 Data Model and Collection

System monitoring data records the interactions among system resources as system events [48]. Each of the recorded event occurs on a particular host at a particular time, thus exhibiting strong spatial and temporal properties. Existing works have indicated that on most modern operating systems (Windows, Linux and OS X), system resources (system entities) in most cases are files, processes, and network connections [42, 45, 48, 49]. Thus, in our data model, we consider system *entities* as *files*, *processes*, and *network connections*. We define a system event as the interaction among two system entities represented using the triple ⟨*subject, operation, object*⟩, which consists of the initiator of the interaction, the type

Table 1: Representative attributes of system entities

Entity	Attributes
File	Name, Owner/Group, VolID, DataID, etc.
Process	PID, Name, User, Cmd, Binary Signature, etc.
Network Connection	IP, Port, Protocol

of the interaction, and the target of the interaction. Subjects are processes originating from software applications such as Firefox, and objects can be files, processes and network connections. We categorize system events into three types according to their object entities, namely *file events*, *process events*, and *network events*.

Both entities and events have critical security-related attributes (Tables 1 and 2). The attributes of entities include the properties to support various security analyses (e.g., file name, process name, and IP addresses), and the unique identifiers to distinguish entities (e.g., file data ID and process ID). The attributes of events include event origins (i.e., agent ID and start time/end time), operations (e.g., file read/write), and other security-related properties (e.g., failure code). Agent ID refers to the unique ID of the host where the entity/event is observed.

Data Collection: We implement data collection agents for Windows and Linux based on ETW event tracing [13] and the Linux Audit Framework [28]. Tables 1 and 2 show representative attributes of our collected data.

3.2 Data Storage

After the modeling, we store the data in relational databases powered by PostgreSQL [19]. Relational databases come with mature indexing mechanisms and are scalable to massive data. However, even with indexes for speeding up queries, relational databases still face challenges in handling high ingest rates of massive system monitoring data. We next describe how we address these challenges to optimize the database storage.

Time and Space Partitioning: System monitoring data exhibits strong *temporal and spatial properties*: the data collected from different agents is independent from each other, and the timestamps of the collected data increase monotonically. Queries of the data are often specified with a specific time range or a host, or across many hosts within some time interval. Therefore, when storing the data, we partition the data in both the time and the space dimensions: separating groups of agents into table partitions and generating one database per day for the data collected on that day. We build various types of indexes on the attributes that will be queried frequently, such as executable name of process, name of file, source/destination IP of network connection.

Hypertable: For large organizations with hundreds or thousands of machines, we scale the data storage using MPP (massively parallel processing) databases Greenplum [14]. These databases intelligently distribute the storage and search of events and entities based on the spatial and temporal properties of our data model.

Table 2: Representative attributes of system events

Operation	Read/Write, Execute, Start/End, Rename/Delete
Time/Sequence	Start Time/End Time, Event Sequence
Misc.	Subject ID, Object ID, Failure Code

Time Synchronization: We correct potential time drifting of events on agents by applying synchronization protocols like Network Time Protocol (NTP) [17] at the client side, and checking with the clock at the server side.

4 Query Language Design

AIQL is designed to specify three types of attack behaviors: multi-step attacks, dependency tracking of attacks, and abnormal system behaviors. In contrast to previous query languages [7, 22, 23, 25] that focus on the specification of relation joins or graph paths, AIQL uniquely integrates the critical primitives for attack investigation, providing explicit constructs for spatial/temporal constraints, relationship specifications, constraint chaining among system events, and the access to aggregate and historical results in sliding time windows. Grammar 1 shows the representative rules of AIQL.

4.1 Multievent AIQL Query

For multievent queries, AIQL provides explicit language constructs for system events (in a natural format of $\{$*subject-operation-object*$\}$), spatial/temporal constraints, and event relationships.

A Running Example: Query 2 specifies an example system behavior that probes user command history files. Multiple context-aware syntax shortcuts (illustrated in comments) are used, such as attribute inference and omitting unreferenced entity IDs (details are given later).

```
agentid = 1 // unique id of the enterprise host
(at "01/01/2017") // time window
proc p2 start proc p1 as evt1
proc p3 read file[".viminfo" || ".bash_history"] as
     evt2 // .viminfo -> name=.viminfo; omit file ID
with p1 = p3, evt1 before evt2
return p2, p1 //p2 -> p2.exe_name, p1 -> p1.exe_name
sort by p2, p1
```

Query 2: Command history probing

Global Constraints: The global constraint rule ($\langle$*global_cstr*$\rangle$) specifies the constraints for all event patterns (e.g., *agentid* and *time window* in Query 2).

Event Pattern: The event pattern rule ($\langle$*evt_patt*$\rangle$) specifies an event pattern that consists of the subject/object entity ($\langle$*entity*$\rangle$), operation ($\langle$*op_exp*$\rangle$), and optional event ID ($\langle$*evt*$\rangle$). The entity rule ($\langle$*entity*$\rangle$) consists of entity type, optional entity ID, and optional attribute constraints ($\langle$*attr_cstr*$\rangle$). Logical operators ("&&" for AND,

"||" for OR, "!" for NOT) can be used in ⟨*op_exp*⟩ and ⟨*attr_cstr*⟩ to form complex expressions. The optional time window rule (⟨*twind*⟩) further narrows down the search for the event pattern. Common time formats (US formats and ISO 8601) and granularities are supported.

```
⟨aiql⟩          ::= ⟨multievent⟩ | ⟨dependency⟩
⟨multievent⟩    ::= (⟨global_cstr⟩)* (⟨m_query⟩)+
⟨dependency⟩    ::= (⟨global_cstr⟩)* ⟨d_query⟩
⟨global_cstr⟩   ::= ⟨cstr⟩ | '(' ⟨twind⟩ ')' | ⟨slide_wind⟩
⟨twind⟩         ::= 'from' ⟨datetime⟩ 'to' ⟨datetime⟩ | ...
⟨slide_wind⟩    ::= ⟨wind_length⟩ ⟨wind_step⟩
Multi-event query:
⟨m_query⟩       ::= ⟨evt_patt⟩+ ⟨evt_rel⟩? ⟨return⟩ ⟨filter⟩?
⟨evt_patt⟩      ::= ⟨entity⟩ ⟨op_exp⟩ ⟨entity⟩ ⟨evt⟩? ('(' ⟨twind⟩ ')')?
⟨entity⟩        ::= ⟨entity_type⟩ ⟨e_id⟩ ? ('[' ⟨attr_cstr⟩']')?
⟨attr_cstr⟩     ::= ⟨cstr⟩
                 |  '!'⟨attr_cstr⟩
                 |  ⟨attr_cstr⟩ ('&&' | '||') ⟨attr_cstr⟩
                 |  '(' ⟨attr_cstr⟩ ')'
⟨cstr⟩          ::= ⟨attr⟩ ⟨bop⟩ ⟨val⟩
                 |  '!'? ⟨val⟩
                 |  ⟨attr⟩ 'not'? 'in' '(' ⟨val⟩ (',' ⟨val⟩)* ')'
⟨op_exp⟩        ::= ⟨op⟩
                 |  '!'⟨op_exp⟩
                 |  ⟨op_exp⟩ ('&&' | '||') ⟨op_exp⟩
                 |  '(' ⟨op_exp⟩ ')'
⟨evt⟩           ::= 'as' ⟨evt_id⟩ ('[' ⟨attr_cstr⟩']')?
⟨evt_rel⟩       ::= 'with' ⟨rel⟩ (',' ⟨rel⟩)*
⟨rel⟩           ::= ⟨attr_rel⟩ | ⟨temp_rel⟩
⟨attr_rel⟩      ::= ⟨e_id⟩'.'⟨attr⟩ ⟨bop⟩ ⟨e_id⟩'.'⟨attr⟩
                 |  ⟨e_id⟩ ⟨bop⟩ ⟨e_id⟩
⟨temp_rel⟩      ::= ⟨evt_id⟩ ('before' | 'after' | 'within') ('[' ⟨val⟩'–'⟨val⟩ ⟨timeunit⟩']')? ⟨evt_id⟩
⟨return⟩        ::= 'return' 'count'? 'distinct'? ⟨res⟩ (',' ⟨res⟩)*
⟨res⟩           ::= ⟨e_id⟩('.'⟨attr⟩)?
                 |  ⟨agg_func⟩'(' ⟨res⟩ ')'
                 |  'as' ⟨rename_id⟩
⟨group_by⟩      ::= 'group by' ⟨res⟩ (',' ⟨res⟩)*
⟨filter⟩        ::= 'having' ⟨expr⟩
                 |  'sort by' ⟨attr⟩ (',' ⟨attr⟩)* ('asc' | 'desc')?
                 |  'top' ⟨int⟩
Dependency query:
⟨d_query⟩       ::= (('forward' | 'backward') ':')? (⟨entity⟩ ⟨op_edge⟩)+ ⟨entity⟩ ⟨return⟩ ⟨filter⟩?
⟨op_edge⟩       ::= ('->' | '<-') '[' ⟨op_exp⟩ ']'
```

Grammar 1: Representative BNF grammar of AIQL

Event Attribute and Temporal Relationships: The event relationship rule (⟨*evt_rel*⟩) specifies how multiple event patterns are related. The attribute relationship rule (⟨*attr_rel*⟩) uses attribute values of event patterns to specify their relationships. In Query 2, `p1=p3` (inferred as `p1.id=p3.id`) indicates that two event patterns `evt1` and `evt2` are linked by the same entity. The temporal relationship rule (⟨*temporal_rel*⟩) specifies temporal order ("before", "after", "within") of event patterns. For example, `evt1 before[1-2 minutes] evt2` specifies that `evt1` occurred 1 to 2 minutes before `evt2`.

Event Return and Filters: The event return rule (⟨*return*⟩) retrieves the attributes of the matched events. Constructs such as "count", "distinct", "top", "having", and "sort by" are provided for result manipulation and filtering.

Context-Aware Syntax Shortcuts: AIQL includes language syntax shortcuts to make queries more concise.

- *Attribute inference*: (1) default attribute names will be inferred if users specify only attribute values in an event pattern, or specify only entity IDs in the return clause. We select the most commonly used attributes in security analysis as the default attributes: `name` for files, `exe_name` for processes, and `dst_ip` for networks; (2) `id` will be used as the default attribute if users specify only entity IDs in attribute relationships.
- *Optional ID*: the ID of entity/event can be omitted if it is not referenced in the event relationship clause or the event return clause.
- *Entity ID reuse*: reusing entity IDs in multiple event patterns implicitly means that these event patterns share the same entity.

For example, in Query 2, `".viminfo"`, `return p2`, and `p1 = p3` will be inferred as `name = ".viminfo"`, `return p2.exe_name`, and `p1.id = p3.id`, respectively. Query 2 also omits the file ID in *evt2* since it is not referenced. We can also replace *p3* with *p1* in *evt2* and omit `p1 = p3`.

4.2 Dependency AIQL Query

AIQL provides the dependency syntax that chains constraints and specifies temporal relationships among event patterns, facilitating the specification of dependency tracking of attacks. The syntax specifies a sequence of event patterns in the form of a path, where nodes in the path represent system entities and edges represent operations. The `forward` and `backward` keywords can be used to specify the temporal order of the events on the path: `forward` (`backward`) means the events found by the left-most event pattern occurred earliest (latest).

```
1 (at "01/01/2017")
2 forward: proc p1["%/bin/cp%", agentid = 2] ->[write]
      file f1["/var/www/%info_stealer%"]
3 <-[read] proc p2["%apache%"]
4 ->[connect] proc p3[agentid=3] // tracking across
      host
5 ->[write] file f2["%info_stealer%"]
6 return f1, p1, p2, p3, f2
```

Query 3: Forward tracking for malware ramification

Query 3 shows a forward dependency query in AIQL that investigates the ramification of malware (`info_stealer`), which originates from host h_a (`agentid = 2`) and affects host h_b (`agentid = 3`) through an Apache web server. Lines 2-3 specify that `p1` writes to `f1`, and then `f1` is read by `p2`. Such syntax eliminates the need to repetitively specify the shared entity (i.e., `f1`) in each event pattern. An example result may show that `p3` is the `wget` process that downloads the malicious script from host h_b. The operation `->[connect]` at Line 4 indicates the search will track dependencies of events across hosts.

4.3 Anomaly AIQL Query

AIQL provides the constructs of *sliding time window* with common aggregation functions (e.g., `count`, `avg`, `sum`) to facilitate the specification of frequency-based system behavioral models. Besides, AIQL provides the construct of *history states*, allowing queries to compare frequencies using historical information.

```
1 (at "01/01/2017")
2 window = 1 min
3 step = 10 sec
4 proc p read ip ipp
5 return p, count(distinct ipp) as freq
6 group by p
7 having freq > 2 * (freq + freq[1] + freq[2]) / 3
```

Query 4: Simple moving average for network frequency

Query 4 shows an anomaly query that specifies a 1-minute sliding time window and computes the moving average [44] to detect network spikes (Line 7). AIQL supports the common types of moving averages through built-in functions (SMA, CMA, WMA, EWMA [44]). For example, the computation of EWMA for network frequency with normalized deviation can be expressed as: `(freq - EWMA(freq, 0.9)) / EWMA(freq, 0.9) > 0.2`.

5 Query Execution Engine

The AIQL query execution engine executes the query context generated by the parser and optimizes the query execution by leveraging domain-specific properties of system monitoring data. Optimizing a query with many constraints is a difficult task due to the complexities of joins and constraints [8]. AIQL addresses these challenges by providing explicit language constructs for spatial/temporal constraints and temporal relationships, so that the query engine can directly optimize the query execution by: (1) using event patterns as a basic unit for generating data queries and leveraging attribute/temporal relationships to optimize the search strategy; (2) leveraging the spatial and temporal properties of system monitoring data to partition the data and executing the search in parallel based on the spatial/temporal constraints.

5.1 Query Execution Pipeline

Fig. 3 shows the execution pipeline of a multievent query. Based on the query semantics, for every event pattern, the engine synthesizes a *SQL data query*, which searches the optimized relational databases (Sec. 3.2) for the matched events. The data query scheduler prioritizes the execution of data queries to optimize execution performance (Sec. 5.2). Execution results of each data query are further processed by the executor to perform joins and filtering to obtain the desired results. Note that by weaving all these join and filtering constraints together, the engine could generate a large SQL with many constraints mixed together. Such strategy suffers from indeterministic optimizations due to the large number of constraints and often causes the execution to last for minutes or even hours (Sec 6.2.2). For an input dependency query, the engine compiles it to an equivalent multievent query for execution. For an anomaly query, the engine maintains the aggregate results as historical states and performs the filtering based on the historical states.

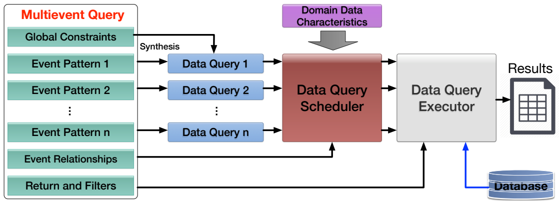


Figure 3: Execution of a multievent AIQL query

5.2 Data Query Scheduler

The data query scheduler in Fig. 3 schedules the execution of data queries. A straightforward scheduling strategy (*fetch-and-filter*) is to: (1) execute data queries separately and store the results of each query in memory; (2) leverage event relationships to filter out results that do not satisfy the constraints. However, this strategy incurs non-trivial computation costs and memory space if some data queries return a large number of results.

Relationship-Based Scheduling: To optimize the execution scheduling of data queries, we leverage two insights based on event relationships: (1) event patterns have different levels of pruning power, and the query engine can prioritize event patterns with more pruning power to narrow the search; (2) if two event patterns are associated with an event relationship, the query engine can execute the data query for the pattern that has more constraints first (likely having more pruning power), and use the execution results to constrain the execution of the other data query.

Algorithm 1 gives the *relationship-based* scheduling:

1. A pruning score is computed for every event pattern based on the number of constraints specified.
2. Event relationships are sorted based on the relationship type (process events and network events are sorted in front of file events) and the sum of the involved event patterns' pruning scores.
3. The main loop processes event relationships returned from the sorted list, executes data queries, and gener-

ates result tuples. The engine executes the data query whose associated event pattern has a higher pruning score first, and leverages existing results to narrow the search scope. To facilitate tuple management, we maintain a map M that stores the mapping from the event pattern ID to the set of event ID tuples that its execution results belong to. As the loop continues, new tuple sets are created and put into M, and old tuple sets are updated, filtered, or merged.

4. After analyzing all event relationships, if there remain unexecuted data queries, these queries are executed and the corresponding results are put into M.
5. The last step is to merge tuple sets in M, so that all event patterns are mapped to the same tuple set that satisfy all constraints.

Algorithm 1: Relationship-based scheduling

```
Input: $n$ data queries: $Q = \{q_i \mid i \leq n, i \in \mathbb{N}^+\}$
       $n$ event patterns: $E = \{e_i \mid i \leq n, i \in \mathbb{N}^+\}$
       $m$ event relationships: $R = \{rel(e_i, e_j)\}$
Output: Event ID tuples that satisfy all constraints
1. $\forall e_i \in E, score(e_i) \xleftarrow{compute} e_i$;
2. $R_{sorted} \xleftarrow{sort} R$;
3. Initialize empty set $Exec$, empty map $M$;
for $rel(e_i, e_j)$ in $R_{sorted}$ do
    if $e_i$ not in $Exec$ and $e_j$ not in $Exec$ then
        // Suppose $score(e_i) \geq score(e_j)$
        $S_i \xleftarrow{execute} q_i$; $Exec.add(e_i)$;        // $S_i$:event ID set
        $S_j \xleftarrow[S_i]{execute} q_j$; $Exec.add(e_j)$;
        $T \leftarrow S_i \times S_j |_{rel(e_i,e_j)}$;   // create tuple set from
        $S_i$ and $S_j$, then filter by $rel(e_i,e_j)$
        $M.put(e_i, T)$; $M.put(e_j, T)$;
    else if Either of $\{e_i, e_j\}$ in $Exec$ then
        // Suppose $e_i$ in $Exec$
        $S_j \xleftarrow[S_i]{execute} q_j$; $Exec.add(e_j)$;
        $T \leftarrow M.get(e_i)$; $T' \leftarrow T \times S_j |_{rel(e_i,e_j)}$;        // update
        tuple set using $S_j$ and $rel(e_i,e_j)$
        $replaceVals(M, T, T')$; $M.put(e_j, T')$;
    else
        $T_i \leftarrow M.get(e_i)$; $T_j \leftarrow M.get(e_j)$;
        if $T_i = T_j$ then
            $T' \leftarrow T_i |_{rel(e_i,e_j)}$;           // filter tuple set
            $replaceVals(M, T_i, T')$;
        else
            $T' \leftarrow T_i \times T_j |_{rel(e_i,e_j)}$;     // merge tuple sets
            $replaceVals(M, T_i, T')$; $replaceVals(M, T_j, T')$;
4. for $e_i \in E$ and $e_i$ not in $Exec$ do
    $S_i \xleftarrow{execute} q_i$; $Exec.add(e_i)$; $M.put(e_i, S_i)$;
5. while $unique(M.values()) > 1$ do
    Pick $T_i$, $T_j$ from $M.values()$, such that $T_i \neq T_j$;
    $T' \leftarrow T_i \times T_j$;                         // merge tuple sets
    $replaceVals(M, T_i, T')$; $replaceVals(M, T_j, T')$;
6. Return $unique(M.values())$;

Function replaceVals (M, T, T')
    Replace all values $T$ stored in $M$ with $T'$;
```

Our empirical results (Sec. 6.3.2 and 6.3.3) demonstrate that the number of constraints work well in approximating the pruning power of event patterns in a broad set of queries, even though they may not accurately represent the size of the results returned by event patterns.

Time Window Partition: The AIQL query engine leverages temporal properties of the data to further speed up the execution of synthesized data queries: the engine partitions the time window of a data query into sub-queries with smaller time windows, and executes them in parallel. Currently, our system splits the time window into days for a query over a multi-day time window.

6 Deployment and Evaluation

We deployed the AIQL system in NEC Labs America comprising 150 hosts (10 servers, 140 employee stations). We performed a series of attacks based on known exploits in the deployed environment and constructed 46 AIQL queries to investigate these attacks, demonstrating the expressiveness of AIQL. To evaluate the effectiveness of AIQL in supporting timely attack investigation, we evaluate the query *efficiency* and *conciseness* against existing systems: PostgreSQL [19], Neo4j [16], Splunk [23]. We also evaluate the efficiency offered by our data query scheduler (Sec. 5.2) in both storage settings: PostgreSQL and Greenplum. In total, our evaluations use 857GB of real system monitoring data (16 days; 2.5 billion events).

6.1 Evaluation Setup

The evaluations are conducted on a database server with an Intel(R) Xeon(R) CPU E5-2660 (2.20GHz), 64GB RAM, and a RAID that supports four concurrent reads/writes. Neo4j databases are configured by importing system entities as nodes and system events as relationships. Greenplum databases are configured to have 5 segment nodes that can effectively leverage the concurrent reads/writes of RAID. For each AIQL query (except anomaly queries), we construct semantically equivalent SQL, Cypher, and Splunk SPL queries. We measure the execution time and the conciseness of each query. Note that we omit the performance evaluation of Splunk since the community version is limited to 500MB per day and the enterprise version is prohibitively expensive ($1,900 per GB). Nevertheless, Splunk's limited support for joins [24] makes it inappropriate for investigating multi-step attack behaviors. Due to the limited expressiveness of SQL and Cypher, we cannot compare the anomaly queries (e.g., Query 5). All queries are available on our *project website [1].*

6.2 Case Study: APT Attack Investigation

We conduct a case study by asking white hat hackers to perform an APT attack in the deployed environment, as

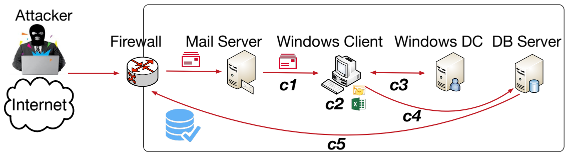


Figure 4: Environmental setup for the APT attack

shown in Fig. 4. Below are the attack steps:

c1 Initial Compromise: The attacker sends a crafted email to the victim. The email contains an Excel file with a malicious macro embedded.

c2 Malware Infection: The victim opens the Excel file through the Outlook mail client and runs the macro, which downloads and executes a malware (CVE-2008-0081 [4]) to open the backdoor to the attacker.

c3 Privilege Escalation: The attacker enters the victim's machine through the backdoor, scans the network ports to discover the IP address of the database, and runs the database cracking tool (`gsecdump.exe`) to obtain the credentials of the user database.

c4 Penetration into Database Server: Using the credentials, the attacker penetrates into the database server and delivers a VBScript to drop another malware, which creates another backdoor to the attacker.

c5 Data Exfiltration: With the access to the database server, the attacker dumps the database content using `osql.exe` and sends the data dump back.

Anomaly Detectors: We deployed two anomaly detectors based on existing solutions [36,52,66]. The first detector is deployed on the database server, which monitors network data transfer and emits alerts when the transfer amount is abnormally large. The second detector is deployed on the Windows client, which monitors process creation and emits alerts when a process starts an unexpected child process. These detectors may produce false positives, and we need tools like AIQL to investigate the alerts before taking any further action.

6.2.1 Attack Investigation Procedure

Our investigation assumes no prior knowledge of the detailed attack steps but merely the detector alerts. We start with these alerts and iteratively compose AIQL queries to investigate the entire attack sequence.

Step c5: We first examine the alerts reported by the database server detector, and identify a suspicious external IP "XXX.129" (obfuscated for privacy). Existing network traffic detectors usually cannot capture the precise process information [50,64]. Thus, we first compose an anomaly AIQL query that computes moving average (SMA3) to find processes which transfer a large amount of data to this suspicious IP.

```
(at "mm/dd/2017") // date (obfuscated)
agentid = xxx // SQL database server (obfuscated)
window = 1 min, step = 10 sec
proc p write ip i[dstip="XXX.129"] as evt
return p, avg(evt.amount) as amt
group by p
having (amt > 2 * (amt + amt[1] + amt[2]) / 3)
```

Query 5: AIQL anomaly query for large file transfer

Query 5 finishes execution within 4 seconds and identifies a suspicious process "sbblv.exe". We then compose a multievent AIQL query to find the data sources for this process (Query 6).

```
(at "mm/dd/2017")
agentid = xx // SQL database server (obfuscated)
proc p1["%sbblv.exe"] read || write file f1 as evt1
proc p1 read || write ip i1[dstip="XXX.129"] as evt2
with evt1 before evt2
return distinct p1, f1, i1, evt1.optype, evt1.access
```

Query 6: Starter AIQL query for *c5*

We identify a suspicious file "BACKUP1.DMP" for `f1` out of the other normal DLL files. We investigate its creation process and find "sqlservr.exe", which is a standard SQL server process with verified signature. Thus, we speculate that the attacker penetrates into the SQL server, dumps the data ("BACKUP1.DMP"), and sends the data back to his host ("XXX.129"). We verify this by checking that "osql.exe" process is started by "cmd.exe" (OSQL utility is often involved in many SQL database attacks). Query 7 gives the complete query for investigating the step *c5*.

```
(at "mm/dd/2017")
agentid = xxx // SQL database server (obfuscated)
proc p1["%cmd.exe"] start proc p2["%osql.exe"] as
     evt1
proc p3["%sqlservr.exe"] write file f1["%backup1.dmp"
     ] as evt2
proc p4["%sbblv.exe"] read file f1 as evt3
proc p4 read || write ip i1[dstip="XXX.129"] as evt4
with evt1 before evt2, evt2 before evt3, evt3 before
     evt4
return distinct p1, p2, p3, f1, p4, i1
```

Query 7: Complete AIQL query for *c5*

Steps c4-c1: The investigation for *c4-c1* is similar to *c5*, including iterative query execution and editing. In total, we constructed 26 multievent queries and 1 anomaly query to successfully investigate the APT attack, touching 119GB of data/422 million events.

6.2.2 Evaluation Results

As we can see, attack investigation is an iterative process that revises queries: (1) latter iterations add more event patterns based on the selected results from the former queries, and (2) 4-5 iterations are needed before finding a complete query with 5-7 event patterns. Thus, *slow response* and *verbose specification* could greatly impede the effectiveness and efficiency of the investigation.

End-to-End Execution Efficiency: Fig. 5 shows the execution time of AIQL queries, SQL queries in PostgreSQL, and Cypher queries in Neo4j. For evaluation

Table 3: Aggregate statistics for case study

Attack Step	# of Queries	# of Evt Patterns	AIQL (s)	PostgreSQL (s)	Neo4j (s)
c1	1	3	3.8	3.1	10.8
c2	8	27	31.0	8038.7	10981.7
c3	2	4	15.9	15.3	3615.6
c4	8	35	61.0	10906.7	8150.6
c5	7	18	58.8	2166.5	4285.4
All	26	87	170.5	21130.3	27044.1

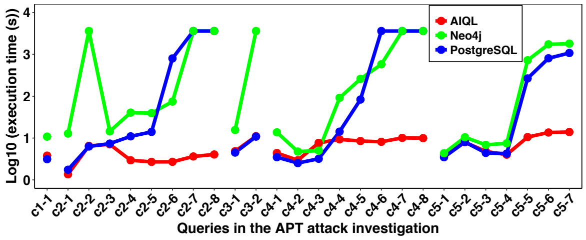


Figure 5: Log10-transformed query execution time

fairness, PostgreSQL and Neo4j databases store the same copies of data and employ the same schema and index designs as AIQL, but they do not employ our domain-specific data storage optimizations such as spatial and temporal partitioning, nor our scheduling optimizations.[2] Table 3 shows aggregate statistics for investigating each attack step, including the number of queries, the number of event patterns, and the total investigation time (second). We observe that: (1) Neo4j generally runs slower than PostgreSQL, due to the lack of support for efficient joins; (2) PostgreSQL and Neo4j become very slow when the query becomes complex and the number of event patterns (hence the required table joins) becomes large. Many large queries in PostgreSQL and Neo4j cannot finish within 1 hour (e.g., c2-7, c2-8, c4-7, c4-8); (3) all AIQL queries finish within 15 seconds, and the performance of the queries grows linearly with the number of event patterns (rather than the exponential growth in PostgreSQL and Neo4j), demonstrating the effectiveness of our domain-specific storage optimizations and query scheduling. (4) the total investigation time is ~5.9 hours for PostgreSQL and ~7.5 hours for Neo4j, which is a significant bottleneck for a timely attack investigation. In contrast, the total investigation time for AIQL is within 3 minutes (124x speedup over PostgreSQL and 157x speedup over Neo4j).

Conciseness: The largest AIQL query is c4-8 with 7 event patterns, 25 query constraints, 109 words, and 463 characters (excluding spaces). The corresponding SQL query contains 77 constraints (3.1x larger), 432 words (4.0x larger), and 2792 characters (6.0x larger). The corresponding Cypher query contains 63 constraints (2.5x larger), 361 words (3.3x larger), and 2570 characters (5.6x larger). As the attack behaviors become more complex, SQL and Cypher queries become verbose with many joins and constraints, posing challenges for constructing the queries for timely attack investigation.

Table 4: Selected malware samples from Virussign

ID	Name	Category
v1	7dd95111e9e100b6243ca96b9b322120	Trojan.Sysbot
v2	425327783e88bb6492753849bc43b7a0	Trojan.Hooker
v3	ee111901739531d6963ab1ee3ecaf280	Virus.Autorun
v4	4e720458c357310da684018f4a254dd0	Virus.Sysbot
v5	7dd95111e9e100b6243ca96b9b322120	Trojan.Hooker

6.3 Performance Evaluation

We evaluate the performance of AIQL in both storage settings (PostgreSQL and Greenplum) by constructing 19 AIQL queries for a broad set of attack behaviors, touching 738GB/2.1 billion events. Particularly, we are interested in the efficiency speedup provided by the AIQL scheduling (Sec. 5.2) in comparison with PostgreSQL scheduling and Greenplum scheduling.

6.3.1 Attack Behaviors

Multi-Step Attack Behaviors: We asked white hat hackers to launch another APT attack using different exploits (details available on [1]). We then constructed 5 AIQL queries for investigating the attack steps (*a1*-*a5*).

Dependency Tracking Behaviors: We performed causal dependency tracking of origins of Chrome update executables (*d1*) and Java update executables (*d2*). We performed forward dependency tracking of the ramification malware `info_stealer` (*d3*).

Real-World Malware Behaviors: We obtained a dataset of free malware samples from VirusSign [33]. We then randomly selected 5 malware samples (Table 4) from the 3 largest categories: *Autorun*, *Sysbot*, and *Hooker*. We executed the 5 selected samples in the deployed environment and constructed AIQL queries by analyzing the accompanied behavior reports [33] (*v1*-*v5*).

Abnormal System Behaviors: We evaluated 6 abnormal system behaviors based on security experts' knowledge: (1) *s1*: command history probing; (2) *s2*: suspicious web service; (3) *s3*: frequent network access; (4) *s4*: erasing traces from system files; (5) *s5*: network access spike; (6) *s6*: abnormal file access. Note that for *s5* and *s6*, we did not construct SQL, Cypher, or Splunk queries, due to their lack of support for sliding window and history state comparison.

6.3.2 Efficiency in PostgreSQL

We select two baselines: (1) PostgreSQL databases that *employ our data storage optimizations (Sec. 3.2)*. Note that this setting is different from the end-to-end efficiency evaluation in Sec. 6.2.2, because here we want to rule out the speedup offered by the data storage component; (2) AIQL with fetch-and-filter scheduling (denoted as AIQL_FF; Sec. 5.2). We measure the execution time of the 19 queries in Sec. 6.3.1.

[2] Fine-grained evaluations of the AIQL scheduling are in Sec. 6.3.

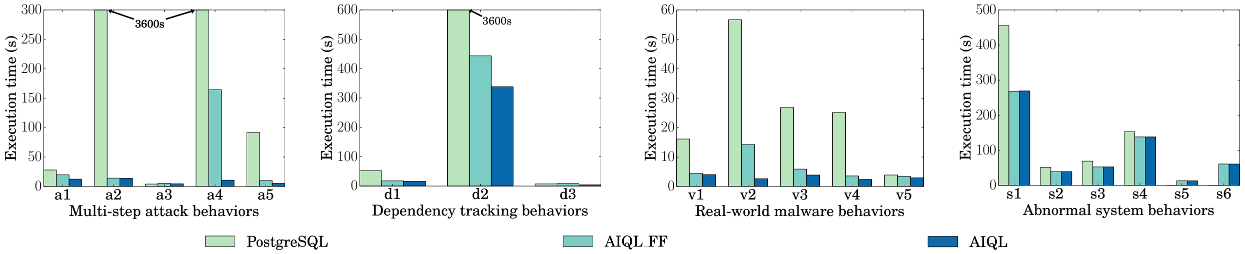


Figure 6: Query execution time of the scheduling employed by PostgreSQL, AIQL_FF, and AIQL (single-node)

Table 5: Conciseness improvement statistics

Metrics	AIQL/SQL	AIQL/ Cypher	AIQL/Splunk SPL
# of constraints	3.0x	2.4x	4.2x
# of words	3.9x	3.1x	3.8x
# of characters	5.3x	4.7x	4.7x

Evaluation Results: Fig. 6 shows the execution time of queries in PostgreSQL, AIQL_FF, and AIQL. We observe that: (1) the scheduling employed by PostgreSQL is inefficient in executing complex queries. In particular, PostgreSQL cannot finish executing *a2*, *a4*, and *d2* within 1 hour; (2) the scheduling employed by AIQL_FF and AIQL is more efficient than PostgreSQL, with 19x and 40x speedup, respectively; (3) the relationship-based scheduling employed by AIQL is more efficient than the fetch-and-filter scheduling employed by AIQL_FF.

6.3.3 Efficiency in Parallel Databases

We compare the performance of AIQL scheduling in the Greenplum storage with the Greenplum scheduling (i.e., running SQLs). As in Sec. 6.3.2, the Greenplum databases also *employ our data storage optimizations*.

Evaluation Results: Fig. 7 shows the execution time of queries in Greenplum and AIQL. We observe that: (1) in most cases, our scheduling in parallel settings achieves a comparable performance as Greenplum scheduling; (2) in certain cases (e.g., *a4*, *d3*), our scheduling is significantly more efficient than Greenplum scheduling; (3) the average speedup over Greenplum is 16x. The results show that without our semantics-aware model, Greenplum distributes the storage of events based on their incoming orders (which is arbitrary). On the contrary, our data model allows Greenplum to evenly distribute events in a host, and achieves more efficient parallel search.

6.4 Conciseness Evaluation

We evaluate the conciseness of queries that express the 19 attack behaviors in Sec. 6.3.1 in three metrics: the number of query constraints, the number of words, and the number of characters (excluding spaces).

Evaluation Results: Fig. 8 shows the conciseness metrics of AIQL, SQL, Neo4j Cypher, and Splunk SPL queries. Table 5 shows the average improvement of AIQL queries over other queries. We observe that AIQL *is the most concise query language* in terms of all three metrics and all attack behaviors: SQL, Neo4j Cypher, and Splunk SPL contain at least 2.4x more constraints, 3.1x more words, and 4.7x more characters than AIQL. In contrast to SQL, Cypher, and SPL which employ lots of joins on tables or nodes, AIQL provides high-level constructs for spatial/temporal constraints, relationship specifications, constraints chaining, and context-aware syntax shortcuts, making the queries much more concise.

7 Discussion

Query Scheduler: Our data query scheduler estimates the pruning score of an event pattern based on its number of constraints. This can be improved by (1) considering the number of records in different hosts and different time periods and (2) constructing a statistical model of constraint pruning power. Additionally, the query scheduler may partition the time window uniformly based on the data volume. Such strategies require further analysis of the domain data statistics to infer the proper data volume for splitting, which we leave for future work.

System Entities and Data Reduction: In the future work, we plan to add registry entries in Windows and pipes in Linux to expand the monitoring scope. We also plan to incorporate more finer granularity system monitoring, such as execution partition [58, 59] and in-memory data manipulations [40, 43]. To handle the increase of data size, we plan to explore more aggressive data reduction techniques in addition to existing solutions [55, 69] to make the system more scalable.

8 Related Work

Security-Related Languages: There also exist domain-specific languages in a variety of security fields that have a well-established corpus of low level algorithms, such as threat descriptions [6, 26, 31], secure overlay networks [46, 56], and network intrusions [35, 39, 65, 68]. These languages provide specialized constructs for their particular problem domain. In contrast to these languages, the novelty of AIQL focuses on querying attack behaviors, including (a) providing specialized constructs

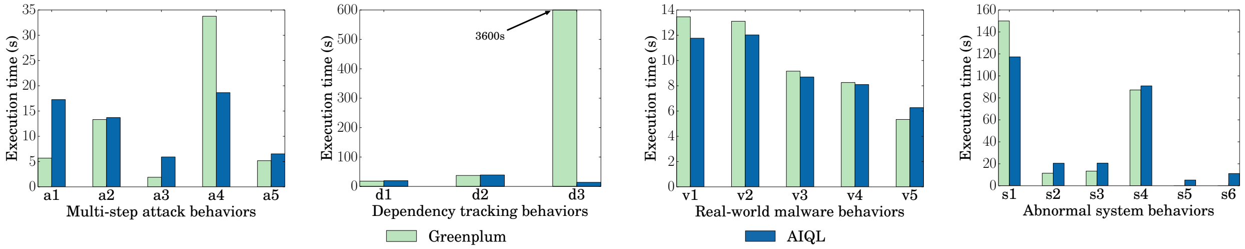
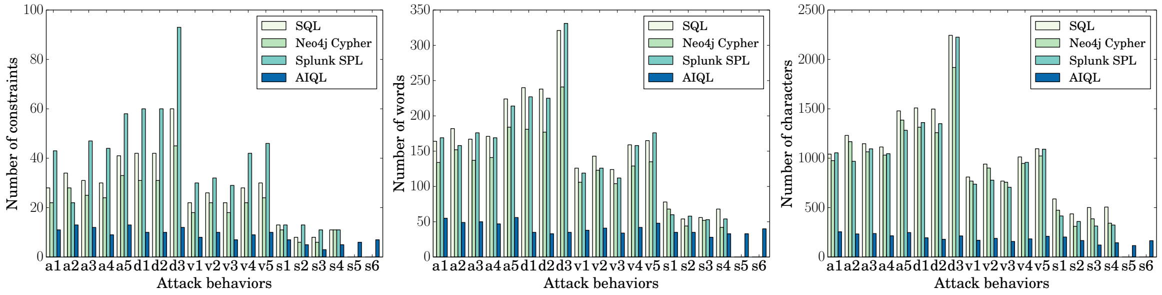


Figure 7: Query execution time of the scheduling employed by Greenplum and AIQL (parallel)



(a) Number of constraints **(b)** Number of words **(c)** Number of characters

Figure 8: Conciseness evaluation of queries written in AIQL, SQL, Neo4j Cypher, and Splunk SPL

for system interaction patterns/relationships and abnormal behaviors; (b) optimizing query execution over system monitoring data. Splunk [23] and Elasticsearch [10] are distributed search and analytics engine for application logs, which provide search languages based on keywords and shell-like piping. However, these systems lack efficient supports for joins and their languages cannot express abnormal behaviors with history states as AIQL. Furthermore, our AIQL can be used to investigate the real-time anomalies detected on the stream of system monitoring data, complementing the stream-based anomaly detection systems [41] for better defense.

Database Query Languages: Relational databases based on SQL [19, 25] and SPARQL [22] provide language constructs for joins, facilitating the specification of relationships among events, but these languages lack constructs for easily chaining constraints among relations (i.e., tables). Graph databases [16] provide language constructs for chaining constraints among nodes in graphs, but these databases lack efficient support for joins. Similarly, NoSQL tools [38] lack efficient supports for joins. Temporal expressions are also introduced to databases [62], and various time-oriented applications are explored [63]. Currently, AIQL focuses on the set of temporal expressions that are frequently used in expressing attack behaviors, which is a subset of the temporal expressions proposed in [62]. More importantly, none of these languages provide constructs to express frequency-based behavioral models with historical results.

System Defense Based on Behavioral Analytics: Existing malware detection has looked at various ways to build behavioral models to capture malware, such as sequences of system calls [67], system call patterns based on data flow dependencies [51], and interactions between benign programs and the operating system [53]. Behavioral analytics have also shown promising results for network intrusion [70, 72] and internal threat detection [60]. These works learn models to detect anomaly or predict attacks, but they do not provide mechanisms for users to perform attack investigation. Our AIQL system fills such gap by allowing security analysts to query historical events for investigating the reported anomalies.

9 Conclusion

We have presented a novel system for collecting attack provenance using system monitoring and assisting timely attack investigation. Our system provides (1) domain-specific data model and storage for scaling the storage and the search of system monitoring data, (2) a domain-specific query language, *Attack Investigation Query Language (*AIQL*)* that integrates critical primitives for attack investigation, and (3) an optimized query engine based on the characteristics of the data and the queries to better schedule the query execution. Compared with existing systems, our AIQL system greatly reduces the cycle time for iterative and interactive attack investigation.

Acknowledgement: This work was partially supported by the National Science Foundation under grants CNS-1553437 and CNS-1409415, Microsoft Research Asia, Jiangsu "Shuangchuang" Talents Program, CCF-NSFOCUS "Kunpeng" Research Fund, and Alipay Research Fund. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] AIQL: Enabling efficient attack investigation from system monitoring data. https://sites.google.com/site/aiqlsystem/.

[2] ANTLR. http://www.antlr.org/.

[3] Apache Flink. https://flink.apache.org/.

[4] CVE-2008-0081. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0081.

[5] CVE-2010-2075. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2075.

[6] Cyber Observable eXpression (CybOX^{TM}). https://cyboxproject.github.io/.

[7] Cypher Query Language. http://neo4j.com/developer/cypher/.

[8] Database performance tuning guide. https://docs.oracle.com/cd/B19306_01/server.102/b14211/.

[9] eBay Inc. To Ask eBay Users To Change Passwords. http://blog.ebay.com/ebay-inc-ask-ebay-users-change-passwords/.

[10] Elasticsearch. https://www.elastic.co/.

[11] The Equifax data breach. https://www.ftc.gov/equifax-data-breach.

[12] Esper. http://www.espertech.com/products/esper.php.

[13] ETW events in the common language runtime. https://msdn.microsoft.com/en-us/library/ff357719(v=vs.110).aspx.

[14] Greenplum. http://greenplum.org/.

[15] Home Depot confirms data breach at U.S., Canadian stores. http://www.npr.org/2014/09/09/347007380/home-depot-confirms-data-breach-at-u-s-canadian-stores.

[16] Neo4j: The world's leading graph database. http://neo4j.com/.

[17] Network Time Protocol (version 3) specification, implementation and analysis. https://tools.ietf.org/html/rfc1305.

[18] OPM government data breach impacted 21.5 million. http://www.cnn.com/2015/07/09/politics/office-of-personnel-management-data-breach-20-million.

[19] PostgreSQL. http://www.postgresql.org/.

[20] Protecting against potentially unwanted programs. https://portal.mcafee.com/documents/Show/2096.

[21] Siddhi. https://github.com/wso2/siddhi.

[22] SPARQL. https://www.w3.org/TR/rdf-sparql-query/.

[23] Splunk. http://www.splunk.com/.

[24] Splunk: joining two searches with common field. https://answers.splunk.com/answers/105469/joining-two-searches-with-common-field.html.

[25] SQL. http://www.iso.org/iso/catalogue_-detail.htm?csnumber=45498.

[26] Structured Threat Information eXpression (STIX^{TM}). http://stixproject.github.io/.

[27] Target data breach incident. http://www.nytimes.com/2014/02/27/business/target-reports-on-fourth-quarter-earnings.html?_r=1.

[28] The Linux audit framework. https://github.com/linux-audit/.

[29] Top 5 causes of sudden network spikes. https://www.paessler.com/press/pressreleases/top_5_causes_-of_sudden_spikes_in_traffic.

[30] Transparent computing. http://www.darpa.mil/program/transparent-computing.

[31] Trusted Automated eXchange of Indicator Information (TAXII^{TM}). https://taxiiproject.github.io/.

[32] Trustwave global security report 2015. https://www2.trustwave.com/rs/815-RFM-693/images/2015_TrustwaveGlobalSecurityReport.pdf.

[33] Virussign. http://www.virussign.com/.

[34] BATES, A., TIAN, D., BUTLER, K. R. B., AND MOYER, T. Trustworthy whole-system provenance for the linux kernel. In *USENIX Security* (2015).

[35] BORDERS, K., SPRINGER, J., AND BURNSIDE, M. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security* (2012).

[36] CHANDOLA, V., BANERJEE, A., AND KUMAR, V. Anomaly detection: A survey. *CSUR 41*, 3 (2009), 15:1–15:58.

[37] CHANDRA, R., KIM, T., SHAH, M., NARULA, N., AND ZELDOVICH, N. Intrusion recovery for database-backed web applications. In *SOSP* (2011).

[38] CHODOROW, K. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media, Inc., 2013.

[39] CUPPENS, F., AND ORTALO, R. Lambda: A language to model a database for detection of attacks. In *RAID* (2000).

[40] DOLAN-GAVITT, B., HODOSH, J., HULIN, P., LEEK, T., AND WHELAN, R. Repeatable reverse engineering with panda. In *PPREW* (2015).

[41] GAO, P., XIAO, X., LI, D., LI, Z., JEE, K., WU, Z., KIM, C. H., KULKARNI, S. R., AND MITTAL, P. SAQL: A stream-based query system for real-time abnormal system behavior detection. In *USENIX Security* (2018).

[42] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The taser intrusion recovery system. In *SOSP* (2005).

[43] GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. R2: An application-level kernel for record and replay. In *OSDI* (2008).

[44] HAMILTON, J. D. *Time series analysis*, vol. 2. Princeton University Press, 1994.

[45] JIANG, X., WALTERS, A., XU, D., SPAFFORD, E. H., BUCHHOLZ, F., AND WANG, Y.-M. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach. In *ICDCS* (2006).

[46] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: Language support for building distributed systems. In *PLDI* (2007).

[47] KIM, T., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Intrusion recovery using selective re-execution. In *OSDI* (2010).

[48] KING, S. T., AND CHEN, P. M. Backtracking intrusions. In *SOSP* (2003).

[49] KING, S. T., MAO, Z. M., LUCCHETTI, D. G., AND CHEN, P. M. Enriching intrusion alerts through multi-host causality. In *NDSS* (2005).

[50] KO, C., RUSCHITZKA, M., AND LEVITT, K. N. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. In *IEEE S&P* (1997).

[51] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X., AND WANG, X. Effective and efficient malware detection at the end host. In *USENIX Security* (2009).

[52] KRUEGEL, C., VALEUR, F., AND VIGNA, G. *Intrusion Detection and Correlation - Challenges and Solutions*, vol. 14 of *Advances in Information Security*. Springer, 2005.

[53] Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., and Kirda, E. Accessminer: Using system-centric models for malware protection. In *CCS* (2010).

[54] Lee, K. H., Zhang, X., and Xu, D. High accuracy attack provenance via binary-based execution partition. In *NDSS* (2013).

[55] Lee, K. H., Zhang, X., and Xu, D. Loggc: Garbage collecting audit log. In *CCS* (2013).

[56] Loo, B. T., Condie, T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., and Stoica, I. Declarative networking: Language, execution and optimization. In *SIGMOD* (2006).

[57] Ma, S., Lee, K. H., Kim, C. H., Rhee, J., Zhang, X., and Xu, D. Accurate, low cost and instrumentation-free security audit logging for windows. In *ACSAC* (2015).

[58] Ma, S., Zhai, J., Wang, F., Lee, K. H., Zhang, X., and Xu, D. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *USENIX Security* (2017).

[59] Ma, S., Zhang, X., and Xu, D. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS* (2016).

[60] Senator, T. E., Goldberg, H. G., Memory, A., Young, W. T., Rees, B., Pierce, R., Huang, D., Reardon, M., Bader, D. A., Chow, E., Essa, I., Jones, J., Bettadapura, V., Chau, D. H., Green, O., Kaya, O., Zakrzewska, A., Briscoe, E., Mappus, R. I. L., McColl, R., Weiss, L., Dietterich, T. G., Fern, A., Wong, W.-K., Das, S., Emmott, A., Irvine, J., Lee, J.-Y., Koutra, D., Faloutsos, C., Corkill, D., Friedland, L., Gentzel, A., and Jensen, D. Detecting insider threats in a real corporate database of computer usage activity. In *KDD* (2013).

[61] Sitaraman, S., and Venkatesan, S. Forensic analysis of file system intrusions using improved backtracking. In *IWIA* (2005).

[62] Snodgrass, R. The temporal query language tquel. *TODS 12*, 2 (1987), 247–298.

[63] Snodgrass, R. T. *Developing Time-oriented Database Applications in SQL*. Morgan Kaufmann Publishers Inc., 2000.

[64] Sommer, R., and Paxson, V. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE S&P* (2010).

[65] Sommer, R., Vallentin, M., De Carli, L., and Paxson, V. Hilti: An abstract execution environment for deep, stateful network traffic analysis. In *IMC* (2014).

[66] Stolfo, S. J., Hershkop, S., Bui, L. H., Ferster, R., and Wang, K. Anomaly detection in computer security and an application to file system accesses. In *ISMIS* (2005).

[67] Sung, A. H., Xu, J., Chavez, P., and Mukkamala, S. Static analyzer of vicious executables (SAVE). In *ACSAC* (2004).

[68] Vallentin, M., Paxson, V., and Sommer, R. Vast: A unified platform for interactive network forensics. In *NSDI* (2016).

[69] Xu, Z., Wu, Z., Li, Z., Jee, K., Rhee, J., Xiao, X., Xu, F., Wang, H., and Jiang, G. High fidelity data reduction for big data security dependency analyses. In *CCS* (2016).

[70] Yen, T.-F., and Reiter, M. K. Traffic aggregation for malware detection. In *DIMVA* (2008).

[71] Yu, S. Understanding the security vendor landscape using the cyber defense matrix, 2016. RSA Conferences.

[72] Zhang, H., Yao, D. D., and Ramakrishnan, N. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *ASIA CCS* (2014).

Application Memory Isolation on Ultra-Low-Power MCUs

Taylor Hardin
Dartmouth College

Ryan Scott
Clemson University

Patrick Proctor
Dartmouth College

Josiah Hester
Northwestern University

Jacob Sorber
Clemson University

David Kotz
Dartmouth College

Abstract

The proliferation of applications that handle sensitive user data on wearable platforms generates a critical need for embedded systems that offer strong security without sacrificing flexibility and long battery life. To secure sensitive information, such as health data, ultra-low-power wearables must isolate applications from each other and protect the underlying system from errant or malicious application code. These platforms typically use microcontrollers that lack sophisticated Memory Management Units (MMU). Some include a Memory Protection Unit (MPU), but current MPUs are inadequate to the task, leading platform developers to software-based memory-protection solutions. In this paper, we present our memory isolation technique, which leverages compiler inserted code and MPU-hardware support to achieve better run-time performance than software-only counterparts.

1 Introduction

Smart watches and smart bands offer novel opportunities for individuals to monitor and control their health, manage a chronic disease, pursue athletic excellence, recover from surgery, or steer their lifestyle toward healthier behaviors. Smart watches can run a variety of apps, including third-party apps installed by the user. However, the battery life for a typical smart watch is about one day, far shorter than the weeks-long battery life typical of single-purpose fitness bands. To balance these trade-offs, some devices (such as the Amulet [10]) seek to achieve the battery life of a closed-source fitness band (like a Fitbit) and the capability to run multiple third-party apps, while retaining strong security properties. These low-energy multi-app wearable platforms employ ultra-low-power microcontrollers (MCUs), with tiny RAM, limited secondary storage, and which lack the hardware-based memory-protection mechanisms – such as Memory Management Units (MMU) – needed to ensure that applications cannot interfere with each other. This makes it difficult to provide long battery life *and* strong security properties that allow multiple third-party apps to coexist.

This work focuses on a fundamental security property: *memory isolation*, which ensures that no application can read, write, or execute memory locations outside its own allocated region, or call functions outside a designated system API. In this paper, we present a novel memory isolation technique, which leverages compiler inserted code and a low-sophistication Memory Protection Unit (MPU) found in many microcontrollers, to achieve better performance than software-only counterparts.

We use the open-source Amulet platform [10] to implement the following isolation methods for comparison: (1) compiler-enforced language limitations (no pointers, no recursion), (2) compiler-inserted run-time memory isolation (address-space bounds verification), and (3) MPU-supported memory isolation (hardware enforced failure). The first option is the approach taken by the Amulet team, which limits the programmer to a subset of C. Pointers are disallowed, and the compiler inserts code for run-time bounds-checking on arrays. In the second approach, we modify the Amulet implementation to allow for pointers and recursion, but our custom compiler inserts code to validate each pointer dereference to ensure the application stays within its bounds. In the third approach we implement a novel combination, in which the OS and compiler coordinate the dynamic assignment of the MPU's limited functionality – and limited compiler-inserted pointer checking – to enable the desired isolation. Finally, we automate this process through an extension to the Amulet Build System. We make the following **contributions**:

1. an analysis of design considerations, including security issues, that enable multiple applications on ultra-low-power wearables, with minimal burden on the programmer or the user;
2. a novel technique, using the limited-function hardware memory protection unit (MPU) found in commodity ultra-low-power microcontrollers, combined

with compile-time analysis of application code, to sandbox application code and memory;

3. a prototype implementation as a refinement of the open-source Amulet platform;
4. an evaluation that compares the performance of the Amulet platform's limited language-based memory-isolation mechanism, a full-featured software-only approach, and a full-featured MPU-assisted mechanism.

2 Background and Related Work

Early operating systems for wireless sensors like TinyOS [14] and others [1, 6, 9] reduced complexity [13], enabled dynamic reprogramming [15], and provided interfaces for concurrent execution [7]. These platforms did not provide memory isolation, nor did they allow installation of multiple third-party applications. As the application space grows, security mechanisms that enable multiprogramming of multi-tenant microcontroller units (MCUs) must be developed. Recent work has explored approaches for memory isolation on microcontrollers.

Some approaches change the language: AmuletOS [10] uses a dialect of ANSI C, termed AmuletC, which disallows pointers and recursion. TockOS [16] writes kernel code in Rust, a type-safe and memory-safe language, and isolates their apps using an MPU. While language modifications can make compile-time analysis easier [17], they tend to limit expressiveness and are rarely enough to ensure complete application isolation.

Language features are often coupled with compiler checks, binary-code rewriting, or system-implemented dynamic checks. For example, AmuletOS has a compiler that inserts run-time bounds-checking code around all array accesses [10]. Deputy [4, 5] enforces type safety at compile time; Harbor [12], built on top of SOS [9], rewrites binary code to check any pointer reference and function call. T-Kernel [8] modifies code at load time to secure application memory. Each of these compile and run-time techniques come with limitations: compile-time techniques depend on language features (or modifications) and clear OS rules, while dynamic checking requires expensive run-time overhead to check memory accesses.

Other systems virtualize the single memory space to isolate applications, like Maté [13], or rely on novel hardware mechanisms such as a Secure Loader hardware unit between the CPU, peripherals, and RAM [11].

Many ultra-low-power MCUs like the MSP430 FRAM series [18] are equipped with a basic Memory Protection Unit, but they have some or all of the following shortcomings: (1) they support too few distinct regions, not enough to sandbox each application; (2) they leave certain segments of memory, like hardware registers or RAM, unprotected; and (3) they have arcane protection boundary rules, because they depend on opaque hardware implementations.

Given all these prior techniques, we see the potential for a *new* approach that leverages the meager capabilities of the new class of MPU, and the lessons learned from years of isolation techniques using software approaches. In this paper, we evaluate the performance of our memory-isolation technique, which leverages compiler-inserted code and MPU-hardware support, against: a language-limited software-based approach (the native Amulet approach [10]), and a full-featured compiler-inserted-check approach.

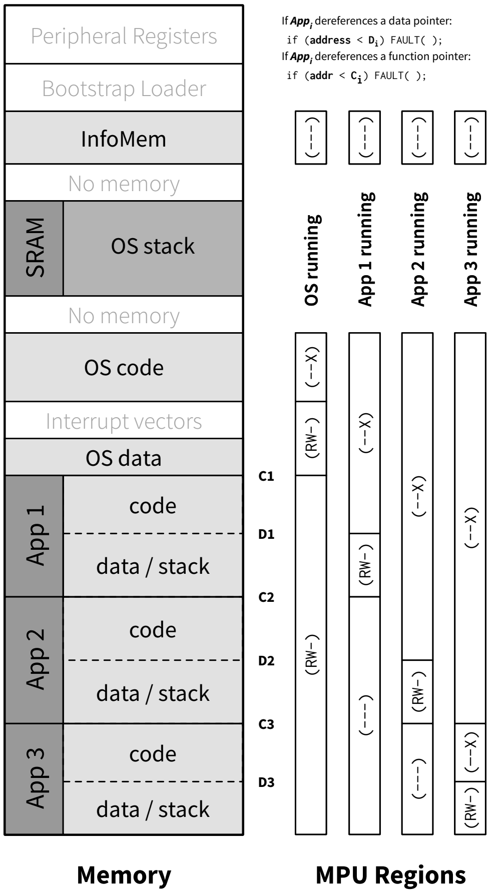
3 System Design

We apply our memory-isolation technique to the latest open-source build of Amulet[1]. Amulet implements memory isolation through compiler-enforced language limitations (no pointers, no recursion, no goto statements and no inline assembly). We remove the most burdensome restrictions by allowing app programmers to use recursion and C pointers (including function pointers) in their code, which reduces the effort to port code to the Amulet and allows developers to write new apps in a customary fashion. In our approach we implement two methods to allow these language features and still ensure memory isolation – use of the memory protection unit (MPU) and compiler-inserted run-time memory isolation.

The Amulet system allows an Amulet user to select a customized mix of applications to run on her Amulet wristband, from a suite of applications developed independently by separate app developers. The Amulet system consists of three core parts – AmuletOS, Amulet Runtime, and the Amulet Firmware Toolchain (AFT). AmuletOS provides the core system services and an event-based scheduler that drives the apps' state machines, delivering events by calling the appropriate event-handler function with parameters representing the details of the event. Amulet Runtime provides a state-machine environment in which all applications run. The Amulet Firmware Toolchain (AFT) [10], analyzes, transforms, merges, and compiles the user's desired applications with the AmuletOS to construct a firmware image for installation on the user's Amulet device.

Amulet devices use a TI MSP430FR5969 MCU, which have a memory protection unit (MPU), with limited capabilities as described in Section 2. The MPU is not a memory-management unit (MMU), nor does it provide full memory protection: it cannot protect all regions of memory (the MPU will not prevent instructions from reading or writing the peripheral registers, InfoMem, SRAM,

[1] The latest open-source release of the Amulet platform can be found at `https://github.com/AmuletGroup/amulet-project`


If *App_i* dereferences a data pointer:

```
if (address < D_i) FAULT( );
```

If *App_i* dereferences a function pointer:

```
if (addr < C_i) FAULT( );
```

Figure 1: Memory diagram of our approach, and MPU regions per application.

or interrupt vectors), and its limited selection of three MPU-controlled segments does not allow us to subdivide memory into the four regions we desire (app code, app data/stack, off-limits memory below the app, and off-limits memory above the app). The MPU only has the ability to protect accesses to memory above the higher app bound but not below the lower app bound. To protect lower memory the compiler inserts a lower bound check. Thus, the MPU memory isolation method consists of configuring the MPU for an app and inserting lower-bound checks, while the compiler inserted (software-only) method mentioned earlier consists of not using the MPU and inserting both an upper and lower memory bound check. Although the MSP430's MPU itself is not sufficient to protect the system and other applications from pointer misuse by a buggy (or malicious) app, it is useful: in our approach, we strategically leverage both the MPU and the compiler to accomplish the necessary protections. This section details the memory map used for MPU, as well as how we handle memory accesses and context switches.

Memory Map: Use of the MPU requires a different memory mapping than in the original Amulet implementation. Figure 1 diagrams our approach. We leverage the SRAM for the AmuletOS stack, the low FRAM for AmuletOS code and data, and the high FRAM for app code and data, grouped by app.[2] Each app's code and data are separated, with its code in lower addresses than its data. The MPU has four segments, of which we can make good use of three.[3]InfoMem, the first segment, is fixed to a certain address range and its configuration can be changed any time by any code. Furthermore, only two boundaries are adjustable: the boundary between segment 1 and 2 and the boundary between segment 2 and 3.

To allow application developers to use C pointers, we leverage previously described MPU hardware. While an app is running, we configure the MPU segments as follows: 0: InfoMem (unused; no access); 1: OS, lower-memory apps, and current-running app's code (execute-only); 2: current-running app's data and stack (read-write only); 3: higher-memory apps (no access).

Consider, as an example, Application 2 in Figure 1. All of the app's code is gathered in one region, all of its data and stack in another region. The MPU configuration triggers a fault if a stray pointer references anything in higher regions (shown as Application 3 in the figure), but the MPU cannot fully protect regions in addresses below the application's code segment.

While the OS is running, we configure the MPU segments as follows: 0: InfoMem (unused; no access); 1: OS code (execute-only); 2: interrupt vectors and OS data (read-write only); 3: apps (read-write only). This configuration allows the AmuletOS to run its own code and, as needed, to manipulate data in both the app and OS regions.

It's important to note an important design change from Amulet as it was originally introduced. The Amulet system uses a single stack – shared by both the OS and the current application. This approach is possible because at most one app runs at any time, so there is no need to retain a stack for non-running apps. It is also possible because app code cannot use pointers, and thus cannot read any memory outside its statically allocated global variables, or outside its current stack frame. If we were to stick with the same single-stack model, we would need to `bzero` the stack region every time we switched apps, lest the new app glean information from the stack tailings left behind by the prior app. We chose instead to allocate a distinct region of memory for each app's stack, removing this cost (and other costs to ensure stack references remain

[2] If the AmuletOS is too large to fit in the low FRAM, it could span the interrupt vectors, but for simplicity we do not show it as such in the diagram.

[3] MPU segment 0 is pinned to the InfoMem, which is only 512 bytes and which we currently do not use. We anticipate using the InfoMem in future revisions, for a return-address stack that protects the return address from stack overflow bugs and attacks.

in-bounds) at the cost of increased memory usage.

That brings us to another important design decision related to security and the application stack. Languages such as C traditionally place a function's return address on the stack, and jump indirectly through that address as part of the function-return instruction. Stack overflows in buggy or malicious code can overwrite that entry on the stack, however, causing the function to return to a different address. We leverage the compiler to insert code to bounds-check the return address before every function return. Furthermore, we place the top of the app stack below the app's data in the app's data/stack segment, and allow the stack to grow downward. The compiler and linker can compute the size of the app's data region, and estimate the maximum stack depth, to ensure the data/stack segment is large enough for the app's needs. If the app overflows its stack, for example by too-deep recursive calls, it will cross an MPU boundary into an execute-only code region and trigger a fault.

Memory accesses: An important role for the runtime system is to handle application faults; when the app attempts an invalid memory access, it jumps to a `FAULT` function to log app-specific information about the fault. At compile time, the AFT uses its transformation tools to verify that the app only calls approved API functions and reads approved system global variables, and to insert code that verifies (at run-time) every pointer dereference before it occurs. Notice that every one of these checks is a simple comparison against a constant, followed by a conditional branch (jump) to the fault-handling code. Because all app code is processed by the AFT, and the app cannot inline any of its own assembly code, the resulting code is guaranteed to check every pointer used by the app.

Context Switches: The AmuletOS provides an API for applications to access utilities and system services. We need to swap MPU configurations and change stacks on each transition, and we need to carefully handle application-provided pointers passed through API calls to the OS. Furthermore, because each app, and the OS, has a separate stack segment, we need to change the stack pointer on every transition between the OS and an app.

AFT Implementation: We extend the AFT to implement the MPU and software-only method checks previously mentioned. These tasks are accomplished by the AFT in a four-phase code analysis. In the first phase, the AFT checks for any still unsupported language features – such as inline assembly and GOTO statements. In addition, the AFT enumerates each memory access and OS API call on an app by app basis. Examination of the application call graph and the stack frame for each function determines the maximum stack size for each app. In the event of recursion, the maximum stack size cannot be determined and the AFT cannot guarantee a large enough stack to prevent overflow. During the second phase, the MPU configuration code and the previously mentioned memory access checks (with placeholder values for app boundaries) are injected into the code. The third phase marks apps with memory section attributes for the linker, as well as injecting the assembly code needed to manipulate the stack pointer. The last phase involves determining the code size of each app, updating the linker script to place each app in high memory (as detailed in Figure 1), and updating the memory access checks from phase two with the correct app boundaries. The AFT completes by recompiling the modified code into the final firmware image.

Operation	No Isolation	Feature Limited	MPU	Software Only
Memory Access	23	41	29	32
Context Switch	90	90	142	98

Table 1: Average cycle count for basic memory isolation operations.

4 Evaluation

In this section we evaluate the costs of application isolation. Our proposed system allows developers to write pure C, instead of a constrained Amulet C, enabling them to more easily write (or port) application code to the Amulet platform. We look at the isolation overhead of a large set of Amulet applications for three methods in Section 4.1, and see that while the overhead of our isolation method is higher than a feature-limited Amulet C, the impact of the overhead on battery lifetime is negligible. In Section 4.2 we describe three benchmark applications, and the tradeoffs they display between computation-intensive and OS-intensive applications.

4.1 Isolation Overhead

We use the Amulet Resource Profiler (ARP) and the ARP-view tool to count the number of memory accesses and context switches per state and transition, per application. Using ARP-view, we can account for the rate of environmental, user, and timer events set by the developer, combine this information with the counted number of memory accesses and context switches, and extrapolate the number of cycles of overhead for isolating applications. We can then convert the estimated cycles into energy cost (in Joules) to estimate the negative impact of isolation on battery lifetime. The results of this experiment are shown in Figure 2 for nine applications that are part of the Amulet platform. These applications comprise thousands of lines of code, and many have been deployed in user studies [2, 3]. **For all applications, isolation using either the MPU or Software Only methods has less than a 0.5% impact on battery lifetime.**

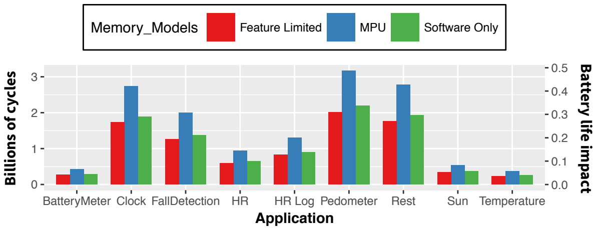


Figure 2: Isolation overhead in billions of cycles per week, and battery lifetime impact percentage for a variety of applications. Gathered using the Amulet Resource Profiler infrastructure.

4.2 Benchmark Applications

We further explore the system overhead of application isolation through several benchmark applications with varying levels of memory accesses. We designed a **Synthetic App** a simple application whose purpose is to test the two fundamental actions that incur memory-protection overheads: *memory accesses* and *context switches*. We then investigate two major functions in our **Activity Detection App**, which correspond to *Activity Case 1* and *Activity Case 2* in Figure 3. These functions have a high number of memory accesses compared to context switches. Finally, we design a **Quicksort App:** an application that runs the quicksort algorithm with a high number of memory accesses and no context switches. Each application was run 200 times and a hardware timer on the MSP430FR5969 MCU was used to measure the time of each iteration (with a precision of 16 cycles).

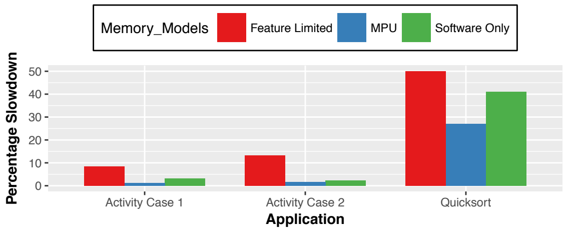


Figure 3: Percentage slowdown for each memory isolation method calculated by comparing them to running apps with no isolation method.

The results from the synthetic app test in Table 1 show that our MPU method had the fastest memory accesses, but the slowest context switches. This result was expected, and validates the simulation results, as our method only requires half the number of bounds checks as the Software Only approach, but incurs extra overhead for reconfiguring the MPU during context switches. Figure 3 further confirms the results from Table 1, which is that our method is the most effective when used for computationally heavy applications.

5 Discussion and Conclusion

In this paper we explore the challenge of memory isolation on ultra-low-power microcontrollers, which offer primitive hardware support for memory protection. Traditional approaches use a range of language limitations, compiler analysis, or dynamic checks (inserted by compiler or other tools); few have leveraged the capabilities of emerging MPUs.

Our solution employs MPU hardware to protect most regions of memory from inappropriate access by application code. Our proof-of-concept implementation (on an Amulet) is limited by the capabilities of the MSP430 MPU, which cannot protect the region below the current app's allocation; thus, the compiler still needs to insert some code for bounds checks – albeit half as many as in the software-only solution. We envision extending our approach to work with more advanced MPUs to further reduce our runtime overheads; MPUs that can protect *all* of memory and support 4 or more regions would negate the need for our compiler-inserted bounds checks. We may also explore more robust error handling techniques, such as restart policies for applications that trigger a memory access fault, or the use of a shadow return-address stack to prevent applications from jumping outside their code bounds.

In conclusion, our exploration shows that (1) it is possible to efficiently support memory isolation without resorting to language limitations, as in the original Amulet approach, and (2) a hybrid approach that leverages compiler-inserted code and MPU-hardware support can provide performance benefits over a software-only approach. While our approach leveraging the MPU was not effective for apps that make frequent API calls, our MPU isolation approach had, for all applications, less than 0.5% impact on battery lifetime.

Acknowledgements

This research results from a research program at the Institute for Security, Technology, and Society, supported by the National Science Foundation under award numbers CNS-1314281, CNS-1314342, CNS-1619970, and CNS-1619950. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the sponsors.

References

[1] BHATTI, S., CARLSON, J., DAI, H., DENG, J., ROSE, J., SHETH, A., SHUCKER, B., GRUENWALD, C., TORGERSON, A., AND HAN, R. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications 10*, 4 (Aug. 2005), 563–579.

[2] BOATENG, G., AND KOTZ, D. StressAware: An app for real-time stress monitoring on the Amulet wearable platform. In *Pro-*

ceedings of the IEEE MIT Undergraduate Research Technology Conference (URTC) (Jan. 2017), IEEE.

[3] BOATENG, G. G. ActivityAware: Wearable system for real-time physical activity monitoring among the elderly. Master's thesis, Dartmouth Computer Science, May 2017. Available as Dartmouth Computer Science Technical Report TR2017-824.

[4] CONDIT, J., HARREN, M., ANDERSON, Z., GAY, D., AND NECULA, G. C. Dependent types for low-level programming. In *European Symposium on Programming* (2007), vol. 4421, Springer, pp. 520–535.

[5] COOPRIDER, N., ARCHER, W., EIDE, E., GAY, D., AND REGEHR, J. Efficient memory safety for TinyOS. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)* (2007), ACM, pp. 205–218.

[6] DUNKELS, A., GRONVALL, B., AND VOIGT, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the Annual IEEE International Conference on Local Computer Networks (LCN)* (2004), pp. 455–462.

[7] DUNKELS, A., SCHMIDT, O., VOIGT, T., AND ALI, M. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2006), SenSys '06, ACM, pp. 29–42.

[8] GU, L., AND STANKOVIC, J. A. T-Kernel: providing reliable os support to wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys)* (2006), pp. 1–14.

[9] HAN, C.-C., KUMAR, R., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. A dynamic operating system for sensor nodes. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2005), ACM, pp. 163–176.

[10] HESTER, J., PETERS, T., YUN, T., PETERSON, R., SKINNER, J., GOLLA, B., STORER, K., HEARNDON, S., FREEMAN, K., LORD, S., HALTER, R., KOTZ, D., AND SORBER, J. Amulet: An energy-efficient, multi-application wearable platform. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)* (Nov. 2016), ACM Press, pp. 216–229.

[11] KOEBERL, P., SCHULZ, S., SADEGHI, A.-R., AND VARADHARAJAN, V. TrustLite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)* (2014), ACM.

[12] KUMAR, R., KOHLER, E., AND SRIVASTAVA, M. Harbor: software-based memory protection for sensor nodes. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN)* (2007), ACM, pp. 340–349.

[13] LEVIS, P., AND CULLER, D. MatÉ: A tiny virtual machine for sensor networks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2002), ACM, pp. 85–95.

[14] LEVIS, P., MADDEN, S., POLASTRE, J., SZEWCZYK, R., WHITEHOUSE, K., WOO, A., GAY, D., HILL, J., WELSH, M., BREWER, E., AND OTHERS. Tinyos: An operating system for sensor networks. *Ambient intelligence 35* (2005), 115–148.

[15] LEVIS, P., PATEL, N., CULLER, D., AND SHENKER, S. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1* (Berkeley, CA, USA, 2004), NSDI'04, USENIX Association, pp. 2–2.

[16] LEVY, A., CAMPBELL, B., GHENA, B., GIFFIN, D. B., PANNUTO, P., DUTTA, D., AND LEVIS, P. Multiprogramming a 64 kb computer safely and efficiently. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (2017), ACM.

[17] SANT'ANNA, F., RODRIGUEZ, N., IERUSALIMSCHY, R., LANDSIEDEL, O., AND TSIGAS, P. Safe system-level concurrency on resource-constrained nodes. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)* (2013), ACM.

[18] TEXAS INSTRUMENTS. Msp430fr5969 16 mhz ultra-low-power microcontroller. `http://ti.com/product/MSP430FR5969`, Oct. 2017.

Peeking Behind the Curtains of Serverless Platforms

Liang Wang [1], Mengyuan Li [2], Yinqian Zhang [2], Thomas Ristenpart[3], Michael Swift[1]

[1]UW-Madison, [2]Ohio State University, [3]Cornell Tech

Abstract

Serverless computing is an emerging paradigm in which an application's resource provisioning and scaling are managed by third-party services. Examples include AWS Lambda, Azure Functions, and Google Cloud Functions. Behind these services' easy-to-use APIs are opaque, complex infrastructure and management ecosystems. Taking on the viewpoint of a serverless customer, we conduct the largest measurement study to date, launching more than 50,000 function instances across these three services, in order to characterize their architectures, performance, and resource management efficiency. We explain how the platforms isolate the functions of different accounts, using either virtual machines or containers, which has important security implications. We characterize performance in terms of scalability, coldstart latency, and resource efficiency, with highlights including that AWS Lambda adopts a bin-packing-like strategy to maximize VM memory utilization, that severe contention between functions can arise in AWS and Azure, and that Google had bugs that allow customers to use resources for free.

1 Introduction

Cloud computing has allowed backend infrastructure maintenance to become increasingly decoupled from application development. Serverless computing (or function-as-a-service, FaaS) is an emerging application deployment architecture that completely hides server management from tenants (hence the name). Tenants receive minimal access to an application's runtime configuration. This allows tenants to focus on developing their functions — small applications dedicated to specific tasks. A function usually executes in a dedicated *function instance* (a container or other kind of sandbox) with restricted resources such as CPU time and memory. Unlike virtual machines (VMs) in more traditional infrastructure-as-a-service (IaaS) platforms, a function instance will be launched only when the function is invoked and is put to sleep immediately after handling a request. Tenants are charged on a per-invocation basis, without paying for unused and idle resources.

Serverless computing originated as a design pattern for handling low duty-cycle workloads, such as processing in response to infrequent changes to files stored on the cloud. Now it is used as a simple programming model for a variety of applications [14, 22, 42]. Hiding resource management from tenants enables this programming model, but the resulting opacity hinders adoption for many potential users, who have expressed concerns about: security in terms of the quality of isolation, DDoS resistance, and more [23, 35, 37, 40]; the need to understand resource management to improve application performance [4, 19, 24, 27, 28, 40]; and the ability of platforms to deliver on performance [10–12, 29–31]. While attempts have been made to shed light on platforms' resource management and security [33, 34], known measurement techniques, as we will show, fail to provide accurate results.

We therefore perform the most in-depth study of resource management and performance isolation to date in three popular serverless computing providers: AWS Lambda, Azure Functions, and Google Cloud Functions (GCF). We first use measurement-driven approaches to partially reverse-engineer the architectures of Lambda and Azure Functions, uncovering many undocumented details. Then, we systematically examine a series of issues related to resource management: how quickly function instances can be launched, function instance placement strategies, function instance reuse, and more. Several security issues are identified and discussed.[1] We further explore how CPU, I/O and network bandwidth are allocated among functions and the ensuing performance implications. Last but not least, we explore whether all resources are properly accounted for, and report on two resource accounting bugs that allow tenants to use extra resources for free. Some highlights of our results include:

- AWS Lambda achieved the best scalability and the lowest coldstart latency (the time to provision a new function instance), followed by GCF. But

[1] We responsibly disclosed our findings to related parties before this paper was made public.

the lack of performance isolation in AWS between function instances from the same account caused up to a 19x decrease in I/O, networking, or coldstart performance.

- Azure Functions used different types of VMs as hosts: 55% of the time a function instance runs on a VM with debased performance.
- Azure had exploitable placement vulnerabilities [36]: a tenant can arrange for function instances to run on the same VM as another tenant's, which is a stepping stone towards cross-function side-channel attacks.
- An accounting issue in GCF enabled one to use a function instance to achieve the same computing resources as a small VM instance at almost no cost.

Many more results are given in the body. We have repeated several measurements in May 2018 and highlight in the paper the improvements the providers have made. We noticed that serverless platforms are evolving quickly; nevertheless, our findings serve as a snapshot of the resource management mechanisms and efficiency of popular serverless platforms, provide performance baselines and design options for developers to build more reliable platforms, and help tenants improve their use of serverless platforms. More generally, our study provides new measurement techniques that are useful for other researchers. Towards facilitating this, we will make our measurement code public and open source.[2]

2 Background

Serverless computing platforms. In serverless computing, an application usually consists of one or more ***functions*** — standalone, small, stateless components dedicated to handle specific tasks. A function is most often specified by a small piece of code written in some scripting language. Serverless computing providers manage the execution environments and backend servers of functions, and allocate resources dynamically to ensure their scalability and availability.

In recent years, many serverless computing platforms have been developed and deployed by cloud providers, including Amazon, Azure, Google, and IBM. We focus on Amazon AWS Lambda, Azure Functions and Google Cloud Functions.[3] In these services, a function is executed in a dedicated container or other type of sandbox with limited resources. We use ***function instance*** to refer to the container/sandbox a function runs on. The resources advertised as available to a function instance varies across platforms, as shown in Table 1. When the function is invoked by requests, one or more function instances (depending on the request volume) will be launched to execute the function. After the function instance(s) have processed the requests and exited or reached the maximum execution time (see "Timeout" in Table 1), the function instance(s) becomes idle. They may be reused to handle subsequent requests to avoid the delay of launching new instances. However, idle function instances can also be suddenly terminated [32]. Each function instance is associated with a non-persistent local disk for temporarily storing data, which will be erased when the function instance is destroyed.

	AWS	**Azure**	**Google**
Memory (MB)	64 * k (k = 2, 3, ..., 24)	1536	128 * k (k = 1, 2, 4, 8, 16)
CPU	Proportional to Memory	Unknown	Proportional to Memory
Language	Python 2.7/3.6 Nodejs 4.3.2/6.10.3 Java 8, and others	Nodejs 6.11.5, Python 2.7, and others	Nodejs 6.5.0
Runtime OS	Amazon Linux	Windows 10	Debian 8*
Local disk (MB)	512	500	> 512
Run native code	Yes	Yes	Yes
Timeout (second)	300	600	540
Billing factor	Execution time Allocated memory	Execution time Consumed memory	Execution time Allocated memory Allocated CPU

Table 1: A comparison of function configuration and billing in three services. (*: We infer the OS version of GCF by checking the help information and version of several Linux tools such as `APT`.)

One benefit of using serverless services is that tenants do not pay for resources consumed when function instances are idle. Tenants are billed based on resource consumption only during execution.[4] In common across platforms is charging for aggregated function execution time across all invocations. Additionally, the price varies depending on the pre-configured function memory (AWS, Google) or the actual consumed memory during invocations (Azure). Google further charges different rates based on CPU speed.

Related work. Many serverless application developers have conducted their own experiments to measure coldstart latency, function instance lifetime, maximum idle time before shut down, and CPU usage in AWS Lambda [10–12, 19, 27, 28, 40]. Unfortunately, their experiments were ad-hoc, and the results may be misleading because they did not control for contention by other instances. A few research papers report on measured performance in AWS. Hendrickson et al. [18] measured request latency and found it had higher latency than AWS Elastic Beanstalk (a platform-as-a-service system). McGrath et al. [34] conducted preliminary measurements on four serverless platforms, and found

[2] `https://github.com/liangw89/faas_measure`

[3] We use AWS, Azure and Google to refer to these services.

[4] Azure Functions offers two types of function hosting plans. *Consumption Plan* manages resources in a serverless-like way while *App Service Plan* is more like "container-as-a-service". We only consider Consumption Plan in this paper.

that AWS achieved better scalability, coldstart latency, and throughput than Azure and Google.

A concurrent study from Lloyd et al. [33] investigated the factors that affect application performance in AWS and Azure. The authors developed a heuristic to identify the VM a function runs on in AWS based on the VM uptime in `/proc/stat`. Our experimental evaluation suggests that their heuristic is unreliable (see §4.5), and that the conclusions they made using it are mostly inaccurate.

In our work, we design a reliable method for identifying instance hosts, and use systematic experiments to inspect resource scheduling and utilization.

3 Methodology

We take the viewpoint of a serverless user to characterize serverless platforms' architectures, performance, and resource management efficiency. We set up vantage points in the same cloud provider region to manage and invoke functions from one or more accounts via official APIs, and leverage the information available to functions to determine important characteristics. We repeated the same experiment under various settings by adjusting function configuration and workloads to determine the key factors that could affect measurement results. In the rest of the paper, we only report on the relevant factors affecting the experiment results.

We integrate all the necessary functionalities and subroutines into a single function that we call a *measurement function*. A measurement function performs two tasks: (1) collect invocation timing and function instance runtime information, and (2) run specified subroutines (e.g., measuring local disk I/O throughput, network throughput) based on received messages. The measurement function collects runtime information via the proc filesystem on Linux (`procfs`), environment variables, and system commands. It also reports on execution start and end time, invocation ID (a random 16-byte ASCII string generated by the function that uniquely identify an invocation), and function configurations to facilitate further analysis.

The measurement function checks the existence of a file named *InstanceID* on the local disk, and if it does not exist, creates this file with a random 16-byte ASCII string that serves as the function instance ID. Since the local disk is non-persistent and has the same lifetime as the associated function instance, the *InstanceID* file will not exist for a fresh function instance, and will not be modified or deleted during the function instance lifetime once created.

The regions for functions were us-east-1, us-central-1, "EAST US" in AWS, Google and Azure (respectively). The vantage points were VMs with at least 4 GB RAM and 2 vCPUs. We used the software recommended by the providers and follow the official instructions to configure the time synchronization service in the vantage points. [5]

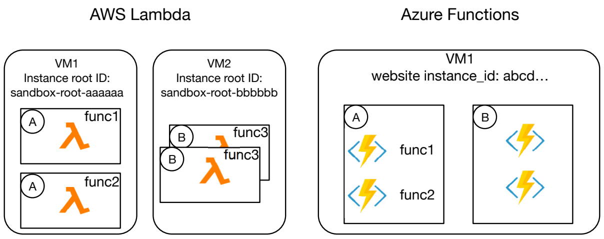


Figure 2: VM and function instance organization in AWS Lambda and Azure Functions. A rectangle represents a function instance. A or B indicates different tenants.

We implemented the measurement function in various languages, but most experiments used Python 2.7 and Nodejs 6.* as the language runtime (the top 2 most popular languages in AWS according to Newrelic [25]). We invoked the functions via synchronous HTTP requests. Most of our measurements were done from July–Dec 2017.

Ethical considerations. We built our measurement functions in a way that should not cause undue burden on platforms or other tenants. In most experiments, the function did no more than collecting necessary information and sleeping for a certain amount of time. Once we discovered performance issues we limited our tests to not DoS other tenants. We only conducted small-scale tests to inspect the security issues but did not further exploit them.

4 Serverless Architectures Demystified

We combine two approaches to infer the architectures of AWS Lambda, Google Cloud Functions, and Azure Functions: (1) reviewing official documents, related online articles and discussions, and (2) measurements — analyzing the data collected from running our measurement functions many times ($>$ 50,000) under varying conditions. This data enables partially reverse engineering the architectures of AWS, Azure, and Google.

4.1 Overview

AWS. A function executes in a dedicated function instance. Our measurements suggest different versions of a function will be treated as distinct and executed in different function instances (we discuss outliers in §5.5). The `procfs` file system exposes global statistics of the underlying VM host, not just a function instance, and contains useful information for profiling runtime,

[5] AWS: `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/set-time.html`; Google: `https://developers.google.com/time/`; Azure does not offer instructions so we use the default NTP servers at `http://www.pool.ntp.org/en/use.html`

(1) Set up N distinct functions $f_1, ..., f_N$ that run the following task upon receiving a `RUN` message: record `/proc/diskstats`, write 20 K – 30 K times to a file (1 byte each time), and record `/proc/diskstats` again.

(2) Invoke each function once without `RUN` message to launch N function instances.

(3) Assuming the instances of $f_1, ..., f_k$ (k instances) share the same instance root ID, invoke $f_1, ..., f_k$ once each with the `RUN` message and examine I/O statistics of each function instance.

Figure 3: I/O-based coresidency test in **AWS**.

identifying host VMs, and more. From `procfs`, we found host VMs mostly have 2 vCPUs and 3.75 GB physical RAM (same as EC2 c4.large instances).

Azure. Azure Functions uses *Function Apps* to organize functions. A function app, corresponding to one function instance, is a container that contains the execution environments for individual functions [5]. The environment variables in the function instance contain some global information about the host VM. The environment variables collected suggest the host VMs can have 1, 2 or 4 vCPUs.

One can create multiple functions in a function app and run them concurrently. In our experiments, we assume that a function app has only one function.

Google. Google isolates and filters information that can be accessed from `procfs`. The files under `procfs` only report usage statistics of the current function instance. Also, many system files and syscalls are obscured or disabled so we cannot get much information about runtime. The `/proc/meminfo` and `/proc/cpuinfo` files suggest a function instance has 2 GB RAM and 8 vCPUs, which we suspect is the configuration for VMs.

4.2 VM identification

Associating function instances with VMs enables us to perform richer analysis. The heuristic for identifying VMs in AWS Lambda proposed by Lloyd et al., though theoretically possible, has never been evaluated experimentally [33]. Therefore, we looked for a more robust method.

AWS. The `/proc/self/cgroup` file has a special entry that we call *instance root ID*. It starts with "sandbox-root-" followed by a 6-byte random string. We found it can be used to reliably identity a host VM. Using the I/O-based coresidency tests (shown in Figure 3), we confirmed that the instances sharing the same instance root ID are on the same VM, as the difference in the total bytes written between two consecutive invocations, for f_i and f_{i+1} respectively, is almost the same as the number of bytes written by f_i. Moreover, we can get the same kernel uptime (or memory usage statistics) from the instances when reading `/proc/uptime` (`/proc/meminfo`) at the same time.

We call the IP obtained via querying IP address lookup tools from an instance *VM public IP*, and the IP obtained from running `uname` command *VM private IP*. Function instances that share the same instance root ID have the same VM public IP and VM private IP.

Azure. The `WEBSITE_INSTANCE_ID` environment variable serves as the VM identifier, according to official documents [6]. We refer to it as *Azure VM ID*. We used Flush-Reload via shared DLLs to verify coresidency of instances sharing the same Azure VM ID [43]. The results suggest Azure VM ID is a robust VM identifier.

Google. We could not find any information enabling us to identify a host. Using I/O-based coresidency did not work as `procfs` contains no global usage statistics. We tried to use performance as a covert-channel (e.g., performing patterned I/O operations in one function instance and detecting the pattern from I/O throughput variation in another) but found this is not reliable, as performance varied greatly (See §6.2).

4.3 Tenant isolation

Prior studies showed that co-located VMs in AWS EC2 allow attacks [36, 38, 41]. With the knowledge of instance-VM relationship, we examined how well tenants' primary resources — function instances — are isolated. We assume that one tenant corresponds to one user account, and only consider VM-level coresidency.

AWS. The functions created by the same tenant will share the same set of VMs, regardless of their configurations and code. The detailed instance placement algorithm will be discussed in §5.1. AWS assigns different VMs to each tenant, since we have never seen function instances from different tenants in the same VM. We conducted a cross-tenant coresidency test to confirm this assumption. The basic principle is similar to Figure 3: in each round, we create a new function under each of the two accounts at the same time, write a random number of bytes in one function, and check the disk usage statistics in another function. We ran this test for 1 week, but found no VM-coresidency of cross-tenant function instances.

Azure. Azure Functions are a part of the Azure App service, in which all tenants share the same set of VMs according to Azure [2]. Hence, tenants in Azure Functions should also share VM resources. A simple test confirmed this assumption: we invoked 500 functions in each of two accounts and found that 30% of function instances were coresident with a function instance from the other account, executing in a total of 120 VMs. Note that as of May 2018, different tenants no longer share the same VMs in Azure. See §5.1 for more details.

4.4 Heterogeneous infrastructure

We found the VMs in all the considered services had a variety of configurations. The variety, likely resulting from infrastructure upgrades, can cause inconsistent function performance. To estimate the fraction of different types of VM in a given service, we examined the configurations of the host VMs of 50,000 unique function instances in each service.

In AWS, we checked the *model_name* and the processor numbers in the `/proc/cpuinfo`, and the *MemTotal* in the `/proc/meminfo`, and found five types of VMs: two E5-2666 vCPUs (2.90 GHZ), two E5-2680 vCPUs (2.80 GHZ), two E5-2676 vCPUs (2.40 GHZ), two E5-2686 vCPUs (2.30 GHZ), and **one** E5-2676 vCPUs. These types account for 59.3%, 37.5%, 3.1%, 0.09% and 0.01% of 20,447 distinct VMs.

Azure shows a greater diversity of VM configurations. The instances in Azure report various vCPU counts: of 4,104 unique VMs, 54.1% use 1 vCPU, 24.6% use 2 vCPUs, and 21.3% use 4 vCPUs. For a given vCPU count, there are three CPU models: two Intel and one AMD. Thus, nine (at least) different types of VMs are being used in Azure. Performance may vary substantially based on what kind of host (more specifically, the number of vCPUs) runs the function. See §6 for more details.

In Google, the *model_name* is always "unknown", but there are 4 unique model versions (79, 85, 63, 45), corresponding to 47.1%, 44.7%, 4.2%, and 4.0% of selected function instances.

4.5 Discussion

Being able to identify VMs in AWS is essential for our measurements. It helps to reduce noise in experiments and get more accurate results. For the sake of comparison, we evaluated the heuristic designed by Lloyd et al. [33]. The heuristic assumes that different VMs have distinct boot times, which can be obtained from `/proc/stat`, and group function instances based on the boot time. We sent 10 – 50 concurrent requests at a time to 1536 MB functions for 100 rounds, used our methodology (instance root ID + IP) to label the VMs, and compared against the heuristic. The heuristic identified 940 VMs as 600 VMs, so 340 (36%) VMs were incorrectly labeled. So, we conclude this heuristic is not reliable.

None of these serverless providers completely hide runtime information from tenants. More knowledge of instance runtime and the backend infrastructure could make finding vulnerabilities in function instances easier for an adversary. In prior studies, `procfs` has been used as a side-channel [9, 21, 46]. In the serverless setting, one actually can use it to monitor the activity of coresident instances; while seemingly harmless, a dedicated adversary might use it as a steppingstone to more sophisticated attacks. Overall, accesses to runtime information, unless necessary, should be restricted for security purposes. Additionally, providers should expose such information in an auditable way, i.e., via API calls, so they are able to detect and block suspicious behaviors.

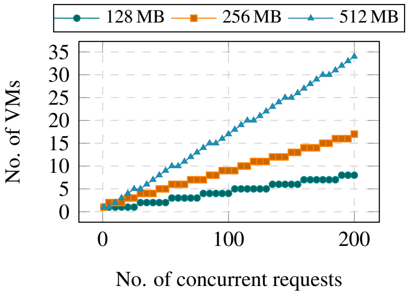


Figure 4: The total number of VMs being used after sending a given number of concurrent requests in **AWS**.

5 Resource Scheduling

We examine how instances and VMs are scheduled in the three serverless platforms in terms of instance coldstart latency, lifetime, scalability, and more.

5.1 Scalability and instance placement

Elastic, automatic scaling in response to changes in demand is a main advertised benefit of the serverless model. We measure how well platforms scale up.

We created 40 measurement functions of the same memory size $f_1, f_2, \ldots, f_{40}$ and invoked each f_i with $5i$ concurrent requests. We paused for 10 seconds between batches of invocations to cope with rate limits in the platforms. All measurement functions simply sleep for 15 seconds and then return. For each configuration we performed 50 rounds of measurements.

AWS. AWS is the best among the three services with regard to supporting concurrent execution. In our measurements, N concurrent invocations always produced N concurrently running function instances. AWS could easily scale up to 200 (the maximum measured concurrency level) fresh function instances.

We observed that **3,328 MB** was the maximum aggregate memory that can be allocated across all function instances on any VM in AWS Lambda. AWS Lambda appears to treat instance placement as a bin-packing problem, and tries to place a new function instance on an existing active VM to maximize *VM memory utilization rates*, i.e., the sum of instance memory sizes divided by 3,328. We invoked a single function with sets of concurrent requests, increasing from 5 to 200 with a step of 5, and recorded the total number of VMs being used after each number of requests. A few examples are shown in Figure 4.

#vCPU	Total	1	2	3	4	>4
1	61.3	16.6	24.6	13.7	4.9	1.5
2	19.5	7.3	7.1	3.3	1.4	0.4
4	19.2	7.6	6.2	3.9	1.3	0.2
All	100	31.5	37.9	20.9	7.6	2.1

Table 5: The average (over 10 runs) probabilities (as percentages) of getting N-way single-account coresidency (for $N \in \{1,2,3,4,\}$ and $N > 4$, when launching 1,000 function instances in **Azure**. Here $N = 1$ indicates no coresidency among the functions.

The number of active VMs are close to the "expected" number if AWS maximizes VM memory utilization. Quantitatively speaking, more than 89% of VMs we got in the test achieved 100% memory utilization. Sending concurrent requests to different functions resulted in the same pattern, indicating placement is agnostic to function code.

In a further test we sent 10 sets of random numbers of concurrent requests to randomly-chosen functions of varied memory sizes over 50 runs. AWS's placement still worked efficiently: the average VM memory utilization rate across VMs in the same run ranged from 84.6% to 100%, with a median of 96.2%.

Azure. Azure documentation states that it will automatically scale up to at most 200 instances for a single Nodejs-based function and at most one new function instance can be launched every 10 seconds [7]. However, in our tests of Nodejs-based functions, we saw at most 10 function instances running concurrently for a single function, no matter how we changed the interval between invocations. All the requests were handled by a small set of function instances. None of the concurrently running instances were on the same VM. So, it appears that Azure does not try to co-locate function instances of the same function on the same VMs.

We conducted a single-account coresidency test to examine how function instances are placed on VMs of different numbers of vCPUs. We invoked 100 different functions from one account at a time until we had 1,000 concurrent, distinct function instances running. We then checked for co-residency, and repeated the entire test 10 times.

We observed at most 8 instances on a single 1/2/4-vCPU VM. Co-resident instances tend to be on 1-vCPU VMs (presumably because there are more 1-vCPU VMs for Azure Functions). We show the breakdown of co-residency results in Table 5. In general, co-residency is undesirable for users wanting many function instances, as contention between instances on low-end VMs will exacerbate performance issues.

We further conducted a cross-account coresidency test in a more realistic scenario where an attacker wants to place her function instances on the same VM with the instances of a target victim. In each round of this test, we launched either 5 or 100 function instances from one account (the *victim*) and 500 simultaneous function instances from another account (the *attacker*). On average, 0.12% (3.82%) of the 500 attacker instances were coresident with the 5 (100) victim instances in each round (10 rounds in total). So, it was possible to achieve cross-tenant coresidency even for a few targets. In the test with 100 victim instances, we were able to obtain up to 5 attacker instances on the same VM. Security implications will be discussed in §5.6.

We repeated the coresidency tests in May 2018 but could not find any cross-tenant coresident instances, even in the test in which we tired 500 victim instances. Therefore, we believe that Azure has fixed the cross-tenant coresidency issue.

Google. Google failed to provide our desired scalability, even though Google claims HTTP-triggered functions will scale to the desired invocation rate quickly [13]. In general, only about half of the expected number of instances, even for a low concurrency level (e.g., 10), could be launched at the same time, while the remainder of the requests were queued.

5.2 Coldstart and VM provisioning

We use *coldstart* to refer to the process of launching a new function instance. For the platform, a coldstart may involve launching a new container, setting up the runtime environment, and deploying a function, which will take more time to handle a request than reusing an existing function instance (*warmstart*). Thus, coldstarts can significantly affect application responsiveness and, in turn, user experience.

For each platform, we created 1,000 distinct functions of the same memory and language and sequentially invoked each of them twice to collect its coldstart and warmstart latency. We use the difference of invocation send time (recorded by the vantage point) and function execution start time (recorded by the function) as an estimation of its coldstart/warmstart latency. As baselines, the median warmstart latency in AWS, Google, and Azure were about 25, 79 and 320 ms (respectively) across all invocations.

AWS. We examine two types of coldstart events: a function instance is launched (1) on a new VM that we have never seen before and (2) on an existing VM. Intuitively, case (1) should have significantly longer coldstart latency than (2) because case (1) may involve starting a new VM. However, we found case (1) was only slightly longer than (2) in general. The median coldstart latency in case (1) was only 39 ms longer than (2) (across all settings). Plus, the smallest VM kernel uptime (from `/proc/uptime`) we found was 132 seconds, indicating that the VM has been launched before the invocation.

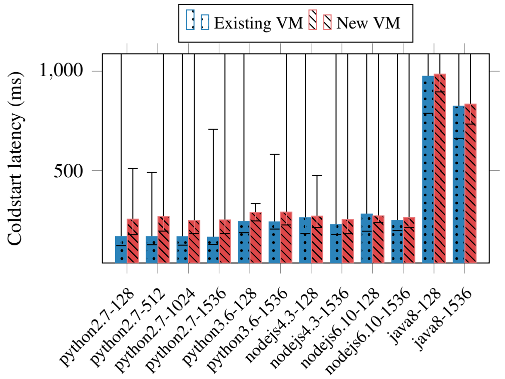


Figure 6: Median coldstart latency with min-max error bars (across 1,000 rounds) under different combinations of function languages and memory sizes in **AWS**. Y-axis is truncated at 1,000 ms.

So, AWS has a pool of ready VMs. The extra delays in case (1) are more likely introduced by scheduling (e.g., selecting a VM) rather than launching a VM.

Our results are consistent with prior observations: function memory and language affect coldstart latency [10], as shown in Figure 6. Python 2.7 achieves the lowest median coldstart latencies (167–171 ms) while Java functions have significantly higher latencies than other languages (824–974 ms). Coldstart latency generally decreases as function memory increases. One possible explanation is that AWS allocates CPU power proportionally to the memory size; with more CPU power, environment set up becomes faster (see §6.1).

A number of function instances may be launched on the same VM concurrently, due to AWS's instance placement strategy. In this case, the coldstart latency increases as more instances are launched simultaneously. For example, launching 20 function instances of a Python 2.7-based function with 128 MB memory on a given VM took 1,321 ms on average, which is about 7 times slower than launching 1 function instance on the same VM (186 ms).

Azure and Google. The median coldstart latency in Google ranged from 110 ms to 493 ms (see Table 7). Google also allocates CPU proportionally to memory, but in Google memory size has greater impact on coldstart latency than in AWS. It took much longer to launch a function instance in Azure, though their instances are always assigned 1.5 GB memory. The median coldstart latency was 3,640 ms in Azure. Anecdotes online [3] suggest that the long latency is caused by design and engineering issues in the platform that Azure is both aware of and working to improve.

Latency variance. We collected the coldstart latencies of 128 MB, Python 2.7 (AWS) or Nodejs 6.* (Google and Azure) based functions every 10 seconds for over 168 hours (7 days), and calculated the median of the coldstart latencies collected in a given hour. The changes of coldstart latency are shown in Figure 8. The coldstart latencies in AWS were relatively stable, as were those in Google (except for a few spikes). Azure had the highest network variation over time, ranging from about 1.5 seconds up to 16 seconds.

Provider-Memory	Median	Min	Max	STD
AWS-128	265.21	189.87	7048.42	354.43
AWS-1536	250.07	187.97	5368.31	273.63
Google-128	493.04	268.5	2803.8	345.8
Google-2048	110.77	52.66	1407.76	124.3
Azure	3640.02	431.58	45772.06	5110.12

Table 7: Coldstart latencies (in ms) in AWS, Google, and Azure using Nodejs 6.* based functions for comparison.

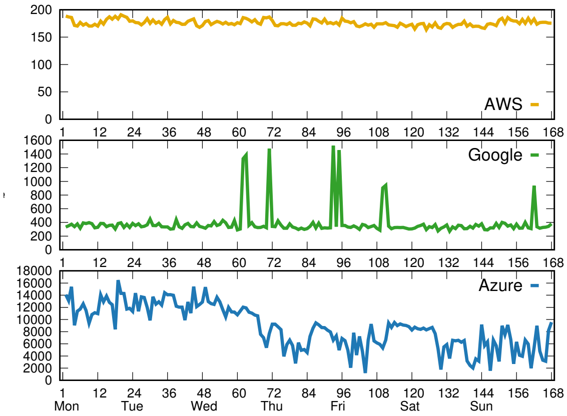


Figure 8: Coldstart latency (in ms) over 168 hours. All the measurements were started at right after midnight on a Sunday. Each data point is the median of all coldstart latencies collected in a given hour. For clarity, the y-axes use different ranges for each service.

We repeated our coldstart measurements in May 2018. We did not find significant changes in coldstart latency in AWS. But, the coldstart latencies became 4x slower on average in Google, probably due to its infrastructure update in February 2018 [15], and 15x better in Azure. This result demonstrates the importance of developing a measurement platform for serverless systems (similar to [39] for IaaS) to do continuous measurements for better performance characterization.

5.3 Instance lifetime

A serverless provider may terminate a function instance even if still in active use. We define the longest time a function instance stays active as *instance lifetime*. Tenants prefer long lifetimes because their applications will be able to maintain in-memory state (e.g., database connections) longer and suffer less from coldstarts.

To estimate instance lifetime, we set up functions of different memory sizes and languages, and invoked

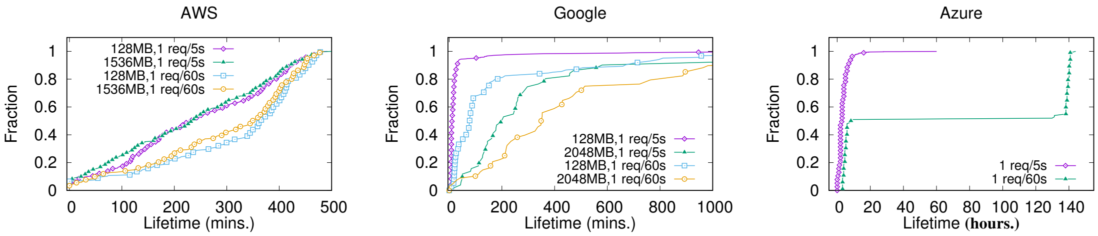


Figure 9: The CDFs of instance lifetime in AWS, Google, and Azure under different memory and request frequency.

them at different frequencies (one request per 5/30/60 seconds). The lifetime of a function instance is the difference between the first time and the last time we saw the instance. We ran the experiment for 7 days (AWS and Google) or longer (Azure) so that we could collect at least 50 lifetimes under a given setting.

In general, Azure function instances have significantly longer lifetimes than AWS and Google as shown in Figure 9. In AWS, the median instance lifetime across all settings was 6.2 hours, with the maximum being 8.3 hours. The host VMs in AWS usually lives longer: the longest observed VM kernel uptime was 9.2 hours. When request frequency increases instance lifetime tends to become shorter. Other factors have little effect on lifetime except in Google, where instances of larger memory tend to have longer lifetimes. For example, when being invoked every five seconds, the lifetimes were 3–31 minutes and 19–580 minutes for 90% of the instances of 128 MB and 2,048 MB memory in Google, respectively. So, for functions with small memory under a heavy workload, Google seems to launch new instances aggressively rather than reusing existing instances. This can increase the performance penalty from coldstarts

5.4 Idle instance recycling

To efficiently use resources, Serverless providers shutdown idle instances to recycle allocated resources (see, e.g., [32]). We define the longest time an instance can stay idle before getting shut down as *instance maximum idle time*. There is a trade-off between long and short idle time, as maintaining more idle instances is a waste of VM memory resources, while fewer ready-to-serve instances cause more coldstarts.

We performed a binary search on the minimum delay t_{idle} between two invocations of the function that resulted in distinct function instances. We created a function, invoked it twice with some delay between 1 and 120 minutes, and determined whether the two requests used the same function instance. We repeated until we identified t_{idle}. We confirmed t_{idle} (to minute granularity) by repeating the measurement 100 times for delays close to t_{idle}.

AWS. An instance could usually stay inactive for at most 27 minutes. In fact, in 80% of the rounds instances were shut down after 26 minutes. When their host VM is "idle", i.e., no active instances on that VM, idle function instances will be recycled the following way: Assuming that the function instances of N functions $f_1, \ldots, f_N$ are coresident on a VM, and k_{f_i} instances are from f_i. For a given function f_i, AWS will shut down $\lfloor k_{f_i}/2 \rfloor$ of the idle instances of f_i every 300 (more or less) seconds until there are two or three instances left, and eventually shut down the remaining instances after 27 minutes (we have tested with $k_{f_i} = 5, 10, 15, 20$). AWS performs these operations to $f_1, \ldots, f_N$ on a given VM independently, and also on individual VMs independently. Function memory or language does not affect maximum idle time.

If there are active instances on the VM, instances can stay inactive for a longer time. We kept one instance active on a given VM by sending a request every 10 seconds and found: (1) AWS still adopted the same strategy to recycle the idle instances of the same function, but (2) somehow idle time was reset for other coresident instances. We observed some idle instances could stay idle in such cases for 1–3 hours.

Azure and Google. In Azure, we could not find a consistent maximum instance idle time. We repeated the experiment several times on different days and found the maximum idle times of 22, 40, and more than 120 minutes. In Google, the idle time of instances could be more than 120 minutes. After 120 minutes, instances remained active in 18% of our experiments.

5.5 Inconsistent function usage

Tenants expect the requests following a function update should be handled by the new function code, especially if the update is security-critical. However, we found in AWS there was a small chance that requests could be handled by an old version of the function. We call such cases *inconsistent function usage*. In the experiment, we sent $k = 1$ or $k = 50$ concurrent requests to a function, and did this again without delay after updating one of the following aspects of the function: IAM role, memory size, environment variables, or function code. For a given setting, we performed these operations for 100 rounds. When $k = 1$, 1%–4% of the tests used an inconsistent function. When there were more associated instances before the update ($k = 50$), 80% of our rounds had at least one inconsistent function. Looking across all tests from all rounds, we found that 3.8% of instances ran an inconsistent function. Examining the cases, we found two situations: (1) AWS launched new instances of the outdated function (2% of all the cases), and (2) AWS reused existing instances of the outdated function. Inconsistent instances never handle more than one request before terminating (note that max execution time is 300 s in AWS), but still, a considerable faction of requests may fail to get desired results.

As we waited for a longer time after the function update to send requests, we found fewer inconsistent cases, and eventually zero cases with a 6-second waiting time. So, we suspect that the inconsistency issues are caused by race conditions in the instance scheduler. The results suggest coordinating function update among multiple function instances is challenging as the scheduler cannot do an atomic update.

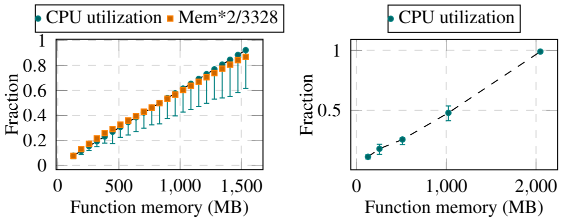
5.6 Discussion

We believe our results motivate further study on designing more efficient instance scheduling algorithms and robust schedulers to further improve VM resource utilization, i.e., to maximize VM memory usage, reduce scheduling latency, and promptly propagate function updates while guaranteeing consistency.

Loading modules or libraries could introduce high latency during coldstart [1, 3]. To reduce coldstart latency, providers might need to adopt more sophisticated library loading mechanisms, for example, using library caching to speed up this process, and resolving the library dependence before deployment and only loading required libraries.

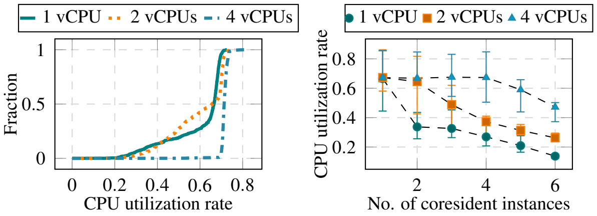
Cross-tenant VM sharing in Azure plus the ability to run arbitrary binaries in the function instance could make applications vulnerable to many kinds of side-channel attacks [16, 17, 20, 45]. We did not examine how well Azure can tackle the potential threats resulting from cross-tenant VM sharing, and leave the actual security vulnerable as an open question.

AWS's bin-packing placement may bring security issues to an application, depending on its design. When a multi-tenant application in Lambda uses IAM roles to isolate its tenants, function instances from different application tenants still share the same VMs. We found two real services that use this pattern: Backand [8] and Zapier [44]. Both allow their tenants to deploy functions in Lambda in some way. We successfully achieved cross-account function coresidency in Backand in just a few tries, while failing in Zapier due to its rate limits and large user base (1 M+). Nevertheless, we could still observe the changes of `procfs` caused by other Zapier tenants' applications, which may admit side-channels [9, 21, 46]. For these multi-tenant applications to isolate their tenants and achieve better security and privacy, AWS may need to provide a finer-grained VM isolation mechanism, i.e., allocating a set of VMs to each IAM role instead of to each account.



(a) AWS (b) Google

Figure 10: The median instance CPU utilization rates with min-max error bars in AWS and Google as function memory increases, averaged across 1,000 instances for a given memory size.



(a) Azure: CPU utilization CDF (b) Azure: CPU vs. coresideny

Figure 11: (a) CDFs of CPU utilization rates of instances (1,000 for each type) and (b) the median CPU utilization rates across a given number of coresident instances (50 rounds) in Azure, with min-max error bars.

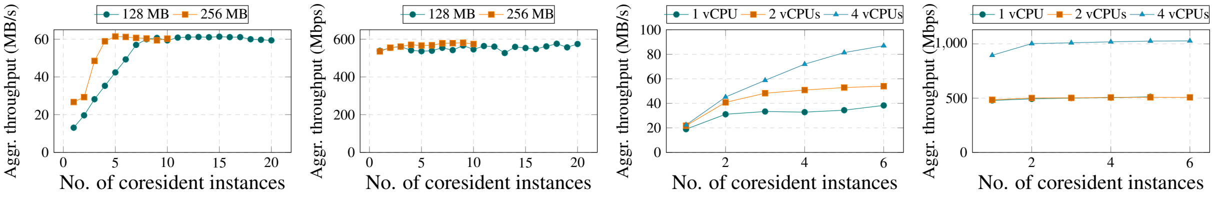
6 Performance Isolation

In this section, we investigate performance isolation. We mainly focus on AWS and Azure, where our ability to achieve coresidency allows more refined measurements. We also present some basic performance statistics for instances in Google that surface seeming contention with other tenants.

6.1 CPU utilization

To measure CPU utilization, our measurement function continuously records timestamps using `time.time()` (Python) or `Date.now()` (Nodejs) for 1,000 ms. The metric *instance CPU utilization rate* is defined as the fraction of the 1,000 ms for which a timestamp was recorded.

AWS. According to AWS, a function instance's CPU power is proportional its pre-configured memory [26]. However, AWS does not give details of how exactly CPU time is allocated to instances. We measured the CPU utilization rates on 1,000 distinct function instances and show the median rate for a given memory size in



(a) AWS: I/O (b) AWS: Network (c) Azure: I/O (d) Azure: Network

Figure 12: Aggregate I/O and network throughput across coresident instances as concurrency level increases. The coresident instances perform the same task simultaneously. The values are the median values across 50 rounds.

Figure 10a. Instances with higher memory get more CPU cycles. The median instance CPU utilization rate increased from 7.7% to 92.3% as memory increased from 128 to 1,536 MB, and the corresponding standard deviations (SD) were 0.7% and 8.7%. When there is no contention from other coresident instances, the CPU utilization rate of an instance can vary significantly, resulting in inconsistent application performance. That said, an upper bound on CPU share is approximated by $2 * m/3328$, where m is the memory size.

We further examine how CPU time is allocated among coresident instances. We let colevel be the number of coresident instances and a colevel of 1 indicates only a single instance on the VM. For memory size m, we selected a colevel in the range 2 to $\lfloor 3328/m \rfloor$. We then measured the CPU utilization rate in each of the coresident instances. Examining the results over 20 rounds of tests, we found that the currently running instances share CPU fairly, since they had nearly the same CPU utilization rate (SD <0.5%). With more coresident instances, each instance's CPU share becomes slightly less than, but still close to $2 * m/3328$ (SD <2.5% in any setting).

The above results indicate that AWS tries to allocate a fixed amount of CPU cycles to an instance based only on function memory.

Azure and Google. Google adopts the same mechanism as AWS to allocate CPU cycles based on function memory [13]. In Google, the median instance CPU utilization rates ranged from 11.1% to 100% as function memory increased. For a given memory size, the standard deviations of the rates across different instances are very low (Figure 10b), ranging from 0.62% to 2.30%.

Azure has a relatively high variance in the CPU utilization rates (14.1%–90%), while the median was 66.9% and the SD was 16%. This is true even though the instances are allocated the same amount of memory. The breakdown by vCPU number shows that the instances on 4-vCPU VMs tend to gain higher CPU shares, ranging from 47% to 90% (Figure 11a). The distributions of utilization rates of instances on 1-vCPU VMs and 2-vCPU VMs are in fact similar; however, when colevel increased, the CPU utilization of instances on 1-vCPU VMs drops more dramatically, as shown in Figure 11b.

6.2 I/O and network

To measure I/O throughput, our measurement functions in AWS and Google used the `dd` command to write 512 KB of data to the local disk 1,000 times (with `fdatasync` and `dsync` flags to ensure the data is written to disk). In Azure, we performed the same operations using a Python script (which used `os.fsync` to ensure data is written to disk). For network throughput measurement, the function used `iperf 3.13` with default configurations to run the throughput test for 10 seconds with different same-region iperf servers, so that iperf server-side bandwidth was not a bottleneck. The iperf servers used the same types of VMs as the vantage points.

AWS. Figure 12 shows aggregate I/O and network throughput across a given number of coresident instances, averaged across 50 rounds. All the coresident instances performed the same measurement concurrently. Though the aggregate I/O and network throughput remains relatively stable, each instance gets a smaller share of the I/O and network resources as colevel increases. When colevel increased from 1 to 20, the average I/O throughput per 128 MB instance dropped by 4x, from 13.1 Mbps to 2.9 Mbps, and network throughput by 19x, from 538.6 MB/s to 28.7 MB/s.

Coresident instances get less share of the network with more contention. We calculate the Coefficient of Variation (CV), which is defined as SD divided by the mean, for each colevel. A higher CV suggests the performance of instances differ more. For 128 MB instances, the CV of network throughput ranged from 9% to 83% across all colevels, suggesting significant performance variability due to contention with coresident instances. In contrast, the I/O performance was similar between instances (CV of 1% to 6% across all colevels). However, the I/O performance is affected by function memory (CPU) for small memory sizes ($\leq$ 512 MB), and therefore the I/O throughput of an instance could degrade more when competing with instances of higher memory.

Azure. In Azure, the I/O and network throughput of an instance also drops as colevel increases, and fluctuates due to contention from other coresident instances. Even more interestingly, resource allocation is differentiated based on what type of VM a function instance happens to be scheduled on. As shown in Figure 12, the 4-vCPU VMs could get 1.5x higher I/O and 2x higher network throughput than the other types of VMs. The 2-vCPU VMs have higher I/O throughput than 1-vCPU VMs, but similar network throughput.

Google. In Google, both the measured I/O and network throughput increase as function memory increases: the median I/O throughput ranged from 1.3 MB/s to 9.5 MB/s, and the median network throughput ranged from 24.5 Mbps to 172 Mbps. The network throughput measured from different instances with the same memory size can vary substantially. For instance, the network throughput measured in the 2,048 MB function instances fluctuated between 0.2 Mbps and 321.4 Mbps. We found two cases: (1) all instances throughputs' fluctuated during a given period of time, irrespective of memory sizes, or (2) a single instance temporarily suffered from degraded throughput. Case (1) may be due to changes in network conditions, while case (2) leads us to suspect that GCF tenants actually share hosts and suffer from resource contention.

6.3 Discussion

AWS and Azure fail to provide proper performance isolation between coresident instances, and so contention can cause considerable performance degradation. In AWS, the fact that they bin-pack function instances from the same account onto VMs means that scaling up a function places the same function on the same VM, resulting in resource contention and prolonged execution time (not to mention a longer coldstart latency). Azure has similar issues, with the additional issue that contention within VMs arises between accounts. The latter also opens up the possibility for cross-tenant degradation of service attacks.

We leave developing new, efficient isolation mechanisms that take the special characteristics of serverless (e.g., frequent instance creation, short-lived instances, and small memory-footprint functions) as considerations for future work.

7 Resource accounting

In the course of our study, we found several resource accounting issues that can be abused by tenants.

Background processes. We found in Google one could execute an external script in the background that continued to run even *after* the function invocation concluded. The script we ran posted a 10 M file every 10 seconds to a server under our control, and the longest time it stayed alive was 21 hours. We could not find any logs of the network activity performed by the background process and were not charged for its resource consumption.[6][7] In contrast, one could run such background script in Azure but Azure logged all the activity. Our observations suggest that: (1) In Azure and Google the function instance execution context will not be frozen after an invocation, as opposed to AWS; and (2) Google does resource accounting via monitoring the Node.js process rather than the entire function instance.

One can exploit the billing issue in Google to run sophisticated tasks at negligible cost. For a function instance with 2 GB memory and 2.4 GHz CPU, one only needs to pay for a few invocations ($0.0000029/100 ms, with 2 M free calls) to get the same computing resources as using a g1-small instance ($0.0257/hour) on Google Cloud Platform.

CPU accounting. In Google, we found there was an 80% chance that a just-launched function instance (of any memory size other than 2,048 MB) could temporally gain more CPU time than expected. Measuring the CPU utilization rates and the completion times of a CPU-intensive task, we confirmed that the instances that one expects to have 8%–58% of the CPU time (see §6) had near 100% of the CPU time, the same as that given to 2,048 MB instances. The instance can retain the CPU resources until the next invocation. Note that if one wants to conduct performance measurements in Google, this issue could introduce a lot of noise (we appropriately controlled for it in previously reported experiments).

8 Conclusion

In this paper, we provided insights into architectures, resource utilization, and the performance isolation efficiency of three modern serverless computing platforms. We discovered a number of issues, raised from either specific design decisions or engineering, with regard to security, performance, and resource accounting in the platforms. Our results surface opportunities for research on improving resource utilization and isolation in future serverless platform designs.

Acknowledgements

The authors thank engineers from Microsoft, Amazon, and Google for their feedback and helpful discussions. This work was supported in part by NSF grants 1558500, 1330308, and 1718084.

[6] Google has a free tier of service, but even after that is used up the background process consumption went unbilled.

[7] We have reported this issue to Google and Google has been working on fixing it as of May 2018.

References

[1] SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association.

[2] Azure app service, virtual machines, service fabric, and cloud services comparison. https://docs.microsoft.com/en-us/azure/app-service/choose-web-site-cloud-service-vm, 2017.

[3] Cold start taking a long time in consumption mode for C# Azure Function. https://github.com/Azure/azure-functions-host/issues/838, 2017.

[4] Consumption plan scaling issues. https://github.com/Azure/azure-webjobs-sdk-script/issues/1206, 2017.

[5] Create your first function in the Azure portal. https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-azure-function, 2017.

[6] Azure runtime environment. https://github.com/projectkudu/kudu/wiki/Azure-runtime-environment, 2017.

[7] Azure Functions scale and hosting. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale, 2017.

[8] Backand. https://www.backand.com/, 2018.

[9] Chen, Q. A., Qian, Z., and Mao, Z. M. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *USENIX Security Symposium* (2014), pp. 1037–1052.

[10] How does language, memory and package size affect cold starts of AWS Lambda? https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76, 2017.

[11] Understanding AWS Lambda performance. https://blog.newrelic.com/2017/01/11/aws-lambda-cold-start-optimization/, 2017.

[12] Understanding AWS Lambda coldstarts. https://www.iopipe.com/2016/09/understanding-aws-lambda-coldstarts/, 2016.

[13] Google Cloud Functions quotas. https://cloud.google.com/functions/quotas, 2017.

[14] Glikson, A., Nastic, S., and Dustdar, S. Deviceless edge computing: extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference* (2017), ACM, p. 28.

[15] Google cloud functions release notes. https://cloud.google.com/functions/docs/release-notes, 2018.

[16] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2016), Springer, pp. 279–299.

[17] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium* (2015), pp. 897–912.

[18] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless computation with openlambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing* (2016), USENIX Association, pp. 33–39.

[19] How long does AWS Lambda keep your idle functions around before a cold start? https://read.acloud.guru/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start-bf715d3b810, 2017.

[20] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: a shared cache attack that works across cores and defies vm sandboxing and its application to AES. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 591–604.

[21] Jana, S., and Shmatikov, V. Memento: Learning secrets from process footprints. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 143–157.

[22] Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., and Recht, B. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 445–451.

[23] Krug, A. Hacking serverless runtimes profiling Lambda, Azure, and more., 2017.

[24] Lambda CPU relative to which instance type? https://forums.aws.amazon.com/message.jspa?messageID=614558, 2014.

[25] AWS Lambda in production. https://blog.newrelic.com/2017/11/21/aws-lambda-state-of-serverless/, 2017.

[26] Configuring Lambda functions. https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html, 2017.

[27] How does proportional CPU allocation work with AWS Lambda? https://engineering.opsgenie.com/how-does-proportional-cpu-allocation-work-with-aws-lambda-41cd44da3cac, 2018.

[28] The occasional chaos of AWS Lambda runtime performance. https://blog.symphonia.io/the-occasional-chaos-of-aws-lambda-runtime-performance-880773620a7e, 2017.

[29] My accidental 35x speed increase of AWS Lambda functions. https://serverless.zone/my-accidental-3-5x-speed-increase-of-aws-lambda-functions-6d95351197f3, 2017.

[30] Comparing AWS Lambda performance when using Node.js, Java, C# or Python. https://read.acloud.guru/comparing-aws-lambda-performance-when-using-node-js-java-c-or-python-281bef2c740f, 2017.

[31] AWS Lambda performance issues. https://stackoverflow.com/questions/43089879/aws-lambda-performance-issues, 2017.

[32] Understanding container reuse in AWS lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, 2014.

[33] Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., and Pallickara, S. Serverless computing: An investigation of factors influencing microservice performance.

[34] McGrath, G., and Brenner, P. R. Serverless computing: Design, implementation, and performance. In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 405–410.

[35] Peterson, E. Serverless security and things that go bump in the night. https://www.infoq.com/presentations/serverless-security, 2017.

[36] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 199–212.

[37] Security and serverless. `https://read.acloud.guru/security-and-serverless-ec52817385c4`, 2017.

[38] VARADARAJAN, V., ZHANG, Y., RISTENPART, T., AND SWIFT, M. M. A placement vulnerability study in multi-tenant public clouds. In *USENIX Security Symposium* (2015), pp. 913–928.

[39] WANG, L., NAPPA, A., CABALLERO, J., RISTENPART, T., AND AKELLA, A. Whowas: A platform for measuring web deployments on iaas clouds. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), ACM, pp. 101–114.

[40] WILLAERT, F. AWS Lambda container lifetime and config refresh. `https://www.linkedin.com/pulse/aws-lambda-container-lifetime-config-refresh-frederik-willaert`, 2016.

[41] XU, Z., WANG, H., AND WU, Z. A measurement study on co-residence threat inside the cloud. In *USENIX Security Symposium* (2015), pp. 929–944.

[42] YAN, M., CASTRO, P., CHENG, P., AND ISHAKIAN, V. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs* (2016), ACM, p. 5.

[43] YAROM, Y., AND FALKNER, K. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium* (2014), pp. 719–732.

[44] Backand. `https://zapier.com/`, 2018.

[45] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 990–1003.

[46] ZHOU, X., DEMETRIOU, S., HE, D., NAVEED, M., PAN, X., WANG, X., GUNTER, C. A., AND NAHRSTEDT, K. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1017–1028.

Soteria: Automated IoT Safety and Security Analysis

Z. Berkay Celik, Patrick McDaniel, and Gang Tan

Department of Computer Science and Engineering
The Pennsylvania State University
{zbc102,mcdaniel,gtan}@cse.psu.edu

Abstract

Broadly defined as the Internet of Things (IoT), the growth of commodity devices that integrate physical processes with digital systems have changed the way we live, play and work. Yet existing IoT platforms cannot evaluate whether an IoT app or environment is safe, secure, and operates correctly. In this paper, we present Soteria, a static analysis system for validating whether an IoT app or IoT environment (collection of apps working in concert) adheres to identified safety, security, and functional properties. Soteria operates in three phases; (a) translation of platform-specific IoT source code into an intermediate representation (IR), (b) extracting a state model from the IR, (c) applying model checking to verify desired properties. We evaluate Soteria on 65 SmartThings market apps through 35 properties and find nine (14%) individual apps violate ten (29%) properties. Further, our study of combined app environments uncovered eleven property violations not exhibited in the isolated apps. Lastly, we demonstrate Soteria on MalIoT, a novel open-source test suite containing 17 apps with 20 unique violations.

1 Introduction

The introduction of IoT devices into public and private spaces has changed the way we live. For example, home automation apps supporting smart devices of thermostats, locks, switches, surveillance systems, and Internet-connected appliances change the way we interact with our living spaces. While these systems have been widely embraced, IoT has also raised concerns about the security and safety of digitally augmented lives [18,21,24,34,36]. IoT apps have access to functions that may put the user or environment at risk, e.g., unlock doors when not at home or create unsafe or damaging conditions by turning off the heat in winter. There has been an increasing amount of recent research exploring IoT security and more broadly environmental safety.

One of the oft-discussed criticisms of IoT is that the software and hardware frameworks do not possess the capability to determine if an IoT device or environment is implemented in a way that is safe, secure, and operates correctly. The SmartThings [37], OpenHab [32], Apple's Homekit [1] provide guidelines and policies for regulating security [2, 31, 43], and related markets provide a degree of internal (hand) vetting of the apps prior to distribution [3,40]. Recent technical community efforts have explored vulnerability analysis within targeted IoT domains [21,30], while others focused on sensitive data leaks and correctness of IoT apps using a range of analyses [8, 17, 25, 45]. However, tools and algorithms for evaluating general safety and security properties within IoT apps and environments are at this time largely absent.

In this paper, we present Soteria[1], a static analysis system for validating whether an IoT app or IoT environment (collection of apps working in concert) adheres to identified safety, security, and functional properties. We exploit existing IoT platforms' sensor-computation-actuator program structures to translate source code of an IoT app into an intermediate representation (IR). Here, the Soteria IR models the app's lifecycle–including app entry points, event handler methods, and call graphs. From this, Soteria uses the IR to perform efficient static analysis extracting a state model of the app; the state model includes its states and transitions. A set of IoT properties is systematically developed, and model checking is used to check that the app (or collection of apps) conforms to those properties. In this work, we make the following contributions:

- We introduce Soteria, a system designed for model checking of IoT apps. Soteria automatically extracts a state model from a SmartThings IoT app and applies model checking to find property violations.
- We used Soteria on 65 different IoT apps (35 apps from the official SmartThings repository and 30 community-contributed third-party apps from the official SmartThings forum) and reveal how safety and security properties are violated.
- We develop an IoT-specific test corpus MalIoT, an open-source repository of 17 flawed apps that containing an array of safety and security violations.

[1] Soteria is the goddess in Greek mythology preserving from harm.

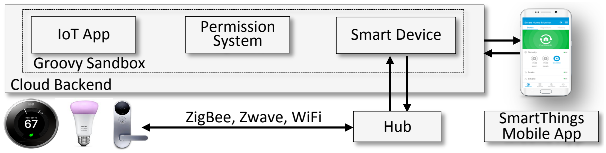


Figure 1: The architecture of SmartThings IoT platform.

2 Background

IoT platforms provide a software stack used to develop applications that monitor and control IoT devices.[2] For example, Fig. 1 shows the three components of the Samsung's SmartThings Platform: a hub, apps, and the cloud backend [40]. The hub controls the communication between connected devices, cloud back-end, and mobile apps. Apps are developed in the Groovy language (a dynamic, object-oriented language) and executed in a Kohsuke sandboxed environment. The cloud backend creates software proxies called SmartDevices that act as a conduit for physical devices, as well as run the apps.

The permission system in SmartThings allows a developer to specify devices and user inputs required for an app at install time. Devices in SmartThings have capabilities (i.e., permissions) that are composed of *actions* and *events*. Actions represent how to control or actuate device states and events are triggered when device states change. SmartThings apps control one or more devices. Apps subscribe to device events or other pre-defined events such as the icon-clicking event, and an event handler is invoked to handle it, which may lead to further events and actions.

Users can install SmartThings apps either from the market or proprietary system via SmartThings Mobile. In the former, publishing an app in the official market requires the developer to submit the source code of the app for review. Official apps appear in the market after the completion of a lengthy review process [40]. In the latter, organizations can develop an app and make it accessible using the Web IDE. These self-published apps do not receive any official review process and are often shared in the SmartThings official community forum [41].

3 Motivation and Assumptions

Example IoT Applications. We introduce three running examples used throughout for exposition and illustration:
The `Smoke-Alarm` app contains a smoke-detection alarm, a water valve (basement), and a light switch (living room). The app sounds the smoke alarm and turns on the light when smoke is detected; when smoke is detected and a heat level is reached, the app opens the water valve to activate fire sprinklers; finally, it turns off the alarm and closes water valve when smoke is clear. Also, it turns on the light switch when the smoke-detector battery is low.
The `Water-Leak-Detector` app detects a water leak with a moisture sensor and shuts off the main water supply valve in order to prevent any further water damage.

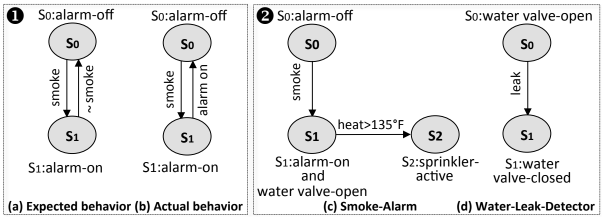


Figure 2: ❶ shows the state models of the expected and actual behavior of the `Smoke-Alarm` app. The app fails because of a bug which halts the alarm when smoke is present. ❷ shows the state models of the `Smoke-Alarm` and `Water-Leak-Detector` apps violating a property when they installed together. The environment fails when the apps interact–the `Water-Leak-Detector` app shuts off water valve when a fire is detected.

The `Thermostat-Energy-Control` app locks the front door and sets the heating thermostat temperature to a predefined value when the user-presence mode is changed (e.g., from the user-away mode to the user-home mode or vice versa). When the energy usage is above a pre-defined consumption threshold, it turns off the thermostat switch.
SOTERIA illustrated. Here we informally illustrate SOTERIA analysis through a single and multi-app example.

Consider the `Smoke-Alarm` app. We first model the app's source code as a transition system. Fig. 2(1a) presents the expected behavior of the smoke alarm; the alarm sounds when smoke is detected and not otherwise. The state model starts from an initial state S_0 and transits to state S_1 when smoke is detected. The state transitions are controlled by the output of the smoke sensor: "smoke-detected" (smoke) and "not detected" (~smoke). Fig. 2(1b) is the actual behavior extracted from the open-source implementation of a smoke alarm (that has a bug). We use SOTERIA to validate the above safety property–i.e., "does the alarm always sound when there is smoke?" To perform this analysis SOTERIA encodes the safety property in temporal logic and verifies it on the model with a symbolic model checker. Naturally, the analysis showed a violation; the actual behavior of the app stops the sound moments after the alarm sounds (the state transition from S_1 to S_0). In this case, users may not hear the short or intermittent alarm with potentially disastrous consequences.

Now consider the situation when both `Smoke-Alarm` and `Water-Leak-Detector` apps are co-located in an environment. Fig. 2(2c) and 2(2d) presents expected behavior of the `Smoke-Alarm` and `Water-Leak-Detector` apps, respectively. Here, we use SOTERIA to validate the property "does the sprinkler system activate when there is a fire?". The model checker revealed that there was a safety violation: the `Water-Leak-Detector` app shuts off the water valve and stops fire sprinklers when it detects water release from sprinklers. In this case, the joint behavior of the otherwise-safe apps leaves users are at risk from fire.

[2] While the SOTERIA approach is largely agnostic to the specific IoT software platform, we focus on Samsung's SmartThings Platform [37].

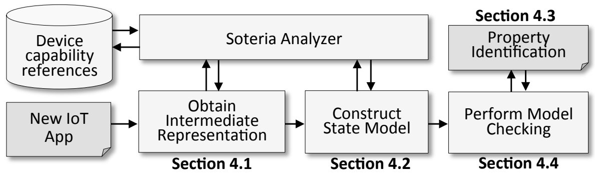


Figure 3: Overview of SOTERIA architecture.

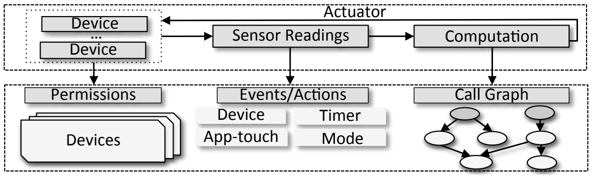


Figure 4: Components of the intermediate representation (IR).

Assumptions and Threat Model. We assume violations can be caused by design flaws or malicious intent. In the latter, the adversary may insert malicious code resulting in insecure or unsafe states, e.g., as seen in attacks on smart light bulbs [36] and home security systems [35]. We do not evaluate adversaries' ability to thwart security measures (e.g., crypto, forged inputs) or explore user privacy, but defer those investigations to future work.

4 SOTERIA

Fig. 3 provides an overview of the four stages of the SOTERIA system analysis. SOTERIA first extracts an intermediate representation (IR) from the source code of an IoT app (Sec. 4.1). The IR is used to model the lifecycle of an app including entry points, event handler methods, and call graphs. Second, SOTERIA uses the IR to extract a state model of the app; the state model includes its states and transitions (Sec. 4.2). Lastly, a set of IoT properties is developed (Sec. 4.3), and model checking is used to check that the app conforms to those properties when running independently or interacting with other apps (Sec. 4.4).

4.1 From Source Code to IR

The first step toward modeling an IoT app is to extract an IR from the app's source code. SOTERIA builds the IR from a framework-agnostic component model, which is comprised of the building blocks of IoT apps, shown in Fig. 4. A broad investigation of existing IoT environments showed that the programming environments could be generalized into three component types: (1) *Permissions* grant capabilities to devices used in an app; (2) *Events/Actions* reflect the association between events and actions: when an event is triggered, an associated action is performed; and (3) *Call graphs* represent the relationship between entry points and functions in an app. The IR has several benefits. First, it allows us to precisely model the app lifecycle as described above. Second, it is used to abstract away parts of the code that are not relevant to property analysis, e.g., `definition` blocks that specify app meta-data and `logger` logging code. Third, it allows efficiently extract the states and state transitions from the implementation (see below). Presented in Fig. 5, we use the `Smoke-Alarm` app to illustrate the use of the IR.

Permissions. When a SmartThings app gets installed or updated, the permissions for devices and user inputs are displayed to the user (and explicitly accepted). The permissions are read-only, and app logic is implemented using the permissions. SOTERIA visits permissions of an app to extract its devices and user inputs. Turning to the IR in Fig. 5, the permission block (lines 1–7) defines: (1) the devices; a smoke detector, a switch, an alarm, a valve, and a battery in the smoke detector; and (2) user input: "thrshld" is used to determine whether the battery level of the smoke detector is low. For each permission, the IR declares a triple following keyword "input". For a device, the triple associates an identifier for the device, called the *device handle*, to its platform-specific device name in order to determine the interface that the device may access. For instance, an app may associate identifier `the_switch` with a switch device, which is in either the "off" or the "on" state. For a user input, the triple in the IR contains the variable name storing the user input, its type, and a tag showing the kind of input such as the user-defined input. In this way, we obtain a complete list of devices and user inputs that an app might access.

Events/Actions. Similar to mobile applications, an IoT app does not have a main method due to its event-driven nature. Apps implicitly define entry points by subscribing events. The event/actions block in an IR is built by analyzing how an app subscribes to events. Each line in the block includes three pieces of information: a device handle, a device event to be subscribed, and an event handler method to be invoked when that event occurs (lines 9–10). Event handler methods are commonly used to take device actions. Therefore, an app may define multiple entry points by subscribing multiple events of a device or devices. Turning to our example, the event of "smoke-detected" state change is associated with an event handler method named `h1()` and the event of "battery" level state change with `h2()`. We also found that events are not limited to device events; we call these *abstract events*: (1) *Timer events*; event-handlers are scheduled to take actions within a particular time or at pre-defined times (e.g., an event-handler is invoked to take actions after a given number of minutes has elapsed or at specific times such as sunset); (2) *App touch events*; for example, some action can be performed when the user clicks on a button in an app; (3) what actions get generated may also depend on *mode events*, which are behavior filters automating device actions. For instance, an app running in "home" mode turns off the alarm and turns on the alarm when it is in the "away" mode. SOTERIA examines all event subscriptions and finds their corresponding event-handler methods; it creates a dummy main method for each entry point.

```
 1: // Permissions block
 2: input (smoke_detector, smokeDetector, type:device)
 3: input (the_switch, switch, type:device)
 4: input (the_alarm, alarm, type:device)
 5: input (the_valve, valve, type:device)
 6: input (the_battery, battery, type:device)
 7: input (thrshld, number, type:user_defined)
 8: // Events/Actions block
 9: subscribe(smoke_detector, "smoke", h1)
10: subscribe(the_battery, "battery", h2)
11: // Entry point
12: h1(){
13:   if(evt.value == "detected") {
14:       the_alarm.siren()
15:       the_valve.open()
16:   }
17:   if(evt.value=="clear"){
18:       the_alarm.off()
19:       the_valve.close()
20:   }
21: }
22: // Entry point
23: h2(){
24:   batteryLevel = p()
25:   if(batteryLevel < thrshld){
26:       the_switch.on()
27:   }
28: }
29: p(){
30:    return the_battery.currentValue("battery")
31: }
```

Figure 5: The IR of Smoke-Alarm app constructed with SOTERIA.

Asynchronously Executing Events. While each event corresponds to a unique event-handler, the sequence of event handler invocations cannot be decided in advance when multiple events happen at the same time. For instance, in our example, there could be a third subscription in the event/actions block that subscribes to the switch-off event to invoke another event handler method. We consider eventually consistent events, which means any time an event handler is invoked, it will finish execution before another event is handled, and the events are handled in the order they are received by an edge device (e.g., a hub). We base our implementation on path-sensitive analysis that analyzes an app's event handlers, which can run in arbitrary sequential order. This analysis is enabled by constructing a separate call graph for each entry point.

Call Graphs. We create a call graph for each entry point that defines an event handler method. Turning to the IR in Fig. 5, we define call graphs for two entry points `h1()` and `h2()` (line 12 and 23). `h1()` invokes `p()` to get the current battery level of the smoke detector. Addressed below, note that these initial graphs are sometimes incomplete because of dynamic method invocations (reflection).

4.2 State Model Extraction

SOTERIA next extracts a state model from the IR model.

Definition of State Models. An IoT app manages one or more devices. Each device has a set of attributes, which are the states of the device. For instance, in the Water-Leak-Detector app, the water sensor has a boolean-typed attribute, whose value signals the "water-detected" or "water-undetected" status. Hence, we naturally model the states in the model from the values of device attributes. IoT apps are event-driven: events such as state changes or user input trigger event handlers, which can in turn change device attributes by invoking device actions. Therefore, by analyzing an IoT app's code, we can add state transitions and label them with events that trigger the transitions (changes to attribute values).

More formally, we define the state model of an IoT app as a triple $(\mathtt{Q}, \Sigma, \delta)$, where Q is a set of states, Σ is a set of transition labels, and δ is a state-transition function that represents labeled transitions between states. We restrict our attention to deterministic state models, as we believe this is a condition for safe operation of IoT devices. In fact, after a state model extracted, SOTERIA reports nondeterministic state models as a safety violation.

Challenges in Extracting State Models. Although it may appear at first glance to be straightforward, extracting state models is fraught with challenges. First, extraction faces state-explosion problem. For instance, a thermostat device may have an integer-discrete or continuous temperature attribute would lead to many different states–adding a state for every possible value in such cases would result in state explosion. To address this, SOTERIA implements a form of property abstraction that collapses states by aggregating attribute values (see Sec. 4.2.1).

A second challenge concerns with model precision. A state model is an abstraction of an app's logic and necessarily has to over-approximate. A sound over-approximation can cause false positives during model checking. One such approximation that caused false positives for an earlier version of SOTERIA was that the labels on transitions were only events and thus too coarse-grained. It turns out that many IoT apps change device states *conditionally*; for example, an app may turn off a switch when the energy consumption is above some threshold and turn on the switch when the energy consumption is below another threshold. For precision, the current version of SOTERIA performs a path-sensitive analysis to extract predicates that guard state changes and adds the predicates as part of state-transition labels. We detail how state transitions are constructed in Sec. 4.2.2.

Finally, the SmartThings platform has a number of idiosyncrasies that SOTERIA's model extraction must address. For instance, SmartThings apps are written in Groovy, a dynamically typed language that supports call by reflection; as another example, SmartThings apps can use special objects for persistent data storage. We will discuss how these issues are addressed in Sec. 4.2.3.

4.2.1 Extracting States

As discussed, states in an app's state model should represent device attribute values. Turning to the Water-Leak-Detector app, this app has two devices: a water sensor and a valve, both of which are represented as Boolean attributes. Therefore, the app's state model has four states: water-detected and valve-closed; water-detected and valve-open; water-undetected and valve-closed; water-undetected and valve-open. The number of possible states of an app is determined by the Cartesian product of the attributes of its device. For instance, an app implementing two devices that have A and B attributes

Algorithm 1: Computing dependence from device's code

```
Input  : ICFG: Inter-procedural control flow graph
Input  : A numerical-valued attribute
Output : Dependence relation dep
$worklist \leftarrow \emptyset$; $done \leftarrow \emptyset$; $dep \leftarrow \emptyset$
for an id in a device action call that sets the attribute at node n do
    $worklist \leftarrow worklist \cup \{(n{:}\ id)\}$
end
while worklist is not empty do
    $(n{:}\ id) \leftarrow worklist.pop()$
    $done \leftarrow done \cup \{(n{:}\ id)\}$
    /* a def of (n: id) at node n′ means a path from n′
       to n exists and on the path there is no other
       assignment to id */
    for a def of (n: id) at node n′ of form id = e and e has only a
      single identifier id′ do
        $worklist \leftarrow worklist \cup (\{(n'{:}\ id')\} \setminus done)$
        $dep \leftarrow dep \cup \{(n{:}\ id, n'{:}\ id')\}$
    end
end
```

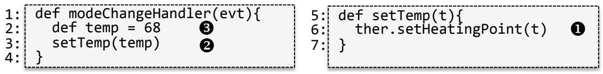
should have states of all pairs $(\mathtt{a},\mathtt{b})$, where $\mathtt{a}\in\mathtt{A}$ and $\mathtt{b}\in\mathtt{B}$.

Identification of Device Attributes. An IoT platform supports many physical devices. Sound model extraction requires identifying the complete set of device attributes. Prior work has used binary instrumentation to observe the runtime behavior of apps to infer the set of device operations used with a particular state [16]. However, this is not an option on some IoT platforms such as SmartThings where app execution is inside proprietary back-ends. Another option would be to use the built-in capability files, which come with devices. The capability file for a device identifies device permissions but not attribute values–and thus do not provide enough information for analysis.

Thus, to identify device attributes, Soteria uses platform-specific device handlers. A device handler is the representation of a physical device in an IoT platform and is responsible for communication between the device and the IoT platform (it is similar to a traditional device driver in an OS). For instance, the switch device handlers in SmartThings [44] and OpenHAB [32] IoT platforms support the "switch on" and "switch off" attributes, and allow apps to incorporate different kinds of switches in the same way. We developed a crawler script, which visits the `status` (for attributes) and `reply` (for actions) code blocks of SmartThings device handlers found in its official GitHub repository [44] and determines a complete set of attributes and actions for devices. We then created our own platform-specific *device capability reference file*, which includes for each device its complete set of attributes and actions. Soteria then uses this file to identify all attributes for those devices used in an app.

Numerical-Valued Device Attributes. Noted above, IoT devices may have attributes with integer or continuous values leading to many states. Returning to the previous `Thermostat-Energy-Control` app, a thermostat with 45 values (50-95 °F) and a power meter with 100 energy levels would lead to (clearly intractable) 4.5K states if a state is added for each combination of attribute values.

Soteria performs property abstraction [5] to reduce the state space. It first performs dependence analysis on an app's source code to identify possible sources for numerical-valued attributes, and then prunes sources using path- and context-sensitivity; the remaining sources are used to construct states in the state model. The Soteria dependence analysis is presented in Algorithm 1 as a worklist-based algorithm. The goal of the algorithm is to identify a set of possible sources that a numerical-valued attribute can take during the execution of an app. The worklist is initialized with identifiers that are used in the arguments of device action calls that change the attribute. The worklist also labels an identifier with node information to uniquely identify the use of an identifier, because the same identifier can be used in multiple locations. The algorithm then takes an entry (n, id) from the worklist and finds a definition for *id* according to the ICFG; if the right-hand side of the definition has a single identifier, the identifier is added to the worklist;[3] furthermore, the dependence between *id* and the right-hand side identifier is recorded in *dep*. For ease of presentation, the algorithm treats parameter passing as inter-procedural definitions.



```
1: def modeChangeHandler(evt){
2:   def temp = 68      ❸
3:   setTemp(temp)      ❷
4: }
```

```
5: def setTemp(t){
6:   ther.setHeatingPoint(t)   ❶
7: }
```

Figure 6: Property abstraction under backward flow analysis.

The dependence analysis is a form of backward taint analysis and produces a set of sources that can affect a change to a numerical-valued attribute. For those sources, Soteria makes them separate states in the state model and adds another state representing the rest of values.

To illustrate, we use a code block of the `Thermostat-Energy-Control` app as an example, shown in Fig. 6. There is a device action call that sets the thermostat to `t` at ❶; so the worklist is initialized to be (6:`t`); for presentation, we use line numbers instead of node numbers to label identifiers. Then, because of the function call at ❷, (3:`temp`) is added to the worklist and the dependence (6:`t`, 3:`temp`) is recorded in *dep*. With this dependence analysis, Soteria computes that the value for `t` has to be 68 °F since `temp` is initialized to be a constant value at ❸. Therefore, the state model has two states for the thermostat: a state when the temperature is equal to 68 °F, and a state when the temperature is not 68 °F; thus, the state space for temperature values is reduced from 45 to 2.

The backward dependence analysis also produces the *dep* relation, through which Soteria constructs paths from identifier initialization points to where device changes happen. For the example in Fig. 6, it produces the path

[3] We found that SmartThings IoT apps most often propagates a developer-defined constant or a user input to places that change device attributes. Occasionally, simple arithmetic is performed; for example, the user input is stored in y, followed by $x = y + 10$, followed by a device attribute change using x. In theory, an IoT app could perform operations like $x = y + z$, where both y and z are user input or defined to be constants; however, we have not encountered this in our evaluation.

❸→❷→❶. Some produced paths by dependence analysis, however, can be infeasible paths. As an optimization, SOTERIA prunes infeasible paths using path- and context-sensitivity. For a path calculated in dependence analysis, it collects the predicates at conditional branches and checks whether the conjunction of those predicates (i.e., the path condition) is always false; if so, the path is infeasible and discarded. This is similar to how symbolic execution prunes paths using path conditions. For instance, if a path goes through two conditional branches and the first branch evaluates $x > 1$ to true and the second evaluates $x < 0$ to true, then it is an infeasible path. SOTERIA does not use a general SMT solver to check path conditions. We found that the predicates used in IoT apps are extremely simple in the form of comparisons between variables and constants (such as $x = c$ and $x > c$); thus, SOTERIA implemented its simple custom checker for path conditions. Furthermore, SOTERIA throws away paths that do not match function calls and returns (using depth-one call-site sensitivity [39]). At the end of the pruning process, we get a set of feasible paths that propagate sources defined by the developer or by user input to device action calls that change the numerical-valued attribute; and then those sources are used to define the states in the model.

4.2.2 Extracting State Transitions

If an event handler changes a device's attributes by actuating the device, it leads to a state transition. By statically analyzing event handlers, SOTERIA computes state transitions and labels them with events. When a water-detected event is generated in the `Water-Leak-Detector` app a handler method closes the valve; by analyzing the handler method, SOTERIA adds a transition with the water-detected event label from state "water-undetected and valve-open" to "water-detected and valve-closed" state.

Labeling Transitions with Predicates. Many device state changes happen in conditional branches; as a result, those state changes occur only when the predicates in the conditional branches hold. To illustrate, consider the source code in Fig. 7 abstracted from the `Thermostat-Energy-Control` app. The app has a conditional branch turning off the switch when energy usage is above a consumption threshold (`above`=50); it turns on the switch when it is below the threshold (`below`=5).

SOTERIA implements a path-sensitive analysis to capture state transitions and predicates that guard transitions. Particularly, it uses *symbolic execution* to perform path exploration on source code and accumulates path conditions during exploration. In detail, it starts the analysis at the entry of an event handler with respect to some initial state, say S_0. Then it performs forward symbolic execution along all paths, and also smartly merges paths following the ESP algorithm [13] (as a way of avoiding path explosion). For a conditional branch with condition b, it evaluates both paths and labels the true path with b and the false path with $\neg b$. If the end states for the true and false branches are the same, then the two paths are merged [13]. On the other hand, if the end states are different for the two paths, they are kept separate for further symbolic execution. SOTERIA throws away infeasible paths in a way similar to that used during property abstraction. At the end of symbolic execution, SOTERIA obtains the set of paths, their end states, and path conditions. For each path, a state transition from the initial state to the end state is added to the state model, and the transition is labeled by the event triggering the event handler and path condition.

```
// Permission block
Input(switch, switch)
Input(power_meter, powerMeter)
// Event/Action block
subscribe(power_meter, power, handler)
// Entry point
handler(){
   above = 50
   below = 5
   power_val = get_power()
   if (power_val > above){
      switch.off()
   }
   if (power_val < below){
      switch.on()
   }
}
get_power(){
   latest_power = power_meter.currentValue("power")
   return latest_power
}
```

Figure 7: The impact of predicates on state transitions in the `Thermostat-Energy-Control` app.

We use the `Thermostat-Energy-Control` app with the initial state of "switch-on" as an illustration of this exploration. SOTERIA explores all paths, and there are two feasible paths at the end, with `currentValue("power")>50` as the path condition of the path that turns off the switch, and `currentValue("power")<5` as the path condition of the path that turns on the switch.

In addition, SOTERIA also tracks the sources of components in predicates that guard state transitions. For predicate `currentValue("power")>50` in the previous example, `currentValue("power")` is obtained from a device state and therefore is labeled as "device-state", while 50 is hardcoded by the developer and therefore is labeled as "developer-defined". In some cases, users can also define part of predicates at install time of an app. For instance, if the threshold value were entered by a user, then SOTERIA would label it as "user-defined". Labeling sources in predicates is useful for precisely stating properties used in model checking. For example, one property says that the alarm must siren when the main door is left open longer than a threshold entered by the user. In this case, there is no property violation if the threshold is not hard-coded into the app by the developer. We detail this in Sec. 4.3.

4.2.3 SmartThings Idiosyncrasies

Platform-specific Interfaces. The SmartThings platform implements a variety of programmer interfaces for an app to obtain device attribute values (for the same value). For instance, the temperature value of a thermostat can be read through the `currentState` or the `currentTemperature` interface (see Listing 1 (lines 1–8). Additionally, we found

Listing 1: Sample code blocks for SmartThings Idiosyncrasies

```
 1 /* A code block of an app using platform-specific interfaces */
 2 subscribe(theMotion, "motion", motionHandler)
 3 subscribe(theThermostat, "thermostat", thermostatHandler)
 4 // different interfaces to get device attribute values
 5 def thermostatHandler() {
 6     def tempAttr = theThermostat.currentState("temperature")
 7     def tempAttr2 = theThermostat.currentThermostat
 8 }
 9 // transitions without explicit event subscriptions
10 def motionHandler(evt) {
11     if (evt.value == "active") { ... }
12     else if (evt.value == "inactive") {...}
13 }
14 /* A code block of an app using call by reflection */
15 //initial state = S0
16 def getMethod(){
17   httpGet("http://url"){ resp ->
18       if(resp.status == 200){
19           name = resp.data.toString()
20       }
21   }
22   "$name"() // dynamic method invocation
23 }
24 // check state transition from S0 to next state in both methods
25 def foo() {...}
26 def bar() {...}
```

that some apps subscribe to all device events instead of specific device events; for example, the `subscribe` interface in Listing 1 (lines 9–13) is used to subscribe to all events of a motion sensor. The event handler then gets an event value as an argument that describes what event it is. We extract precise state models by parsing the event values passed in these interfaces and adding state transitions through those interfaces.

Call by Reflection. The Groovy language supports programming by reflection (using the `GString` feature) [44], which allows a method to be invoked by providing its name as a string. For instance, a Groovy method `foo()` can be invoked by declaring a string `name="foo"` requested from an external server via the `httpGet()` interface and thereafter called by reflection through `$name` (see Listing 1 (lines 14–26)). To handle calls by reflection, Soteria's call graph construction adds all methods in an app as possible call targets, as a safe over-approximation. For the example in Listing 1, Soteria adds both `foo()` and `bar()` to the targets of the call; then it searches for state changes in each method and extracts state transitions.

4.3 Identifying IoT Properties

As many have found in the security and safety communities, identifying the correct set of properties to validate for a given artifact is often a daunting task. In this work and as described below, we use established techniques adapted from other domains to systematically identify a set of properties that exercise Soteria and are representative of the real world needs of users and environments. That being said, we acknowledge in practice that properties are often more contextual and the methods to find them are often more art than science. Hence, we argue that many environments will need to tailor their property discovery process to their specific security and safety needs.

We refer to a property as a system artifact that can be formally expressed via specification and validated on the application model. We extend the use/misuse case requirements engineering [29, 33, 38, 47] to identify IoT properties. This approach derives requirements (properties) by evaluating the connections between 1) *assets* are artifacts that someone places value upon, e.g., a garage door, 2) *functional requirements* define how a system is supposed to operate in normal environment, e.g., when a garage door button is opened, the door opens, and 3) *functional constraints* restrict the use or operation of assets, e.g., the door must open only when an authorized garage-door opener device requests it. We used use/misuse case requirements engineering as a property discovery process on the IoT apps used in our evaluation (See Section 6) and identified 5 general properties (S.1-S.5, see Fig. 8) and 30 application-specific properties (P.1-P.30, see Table 1).

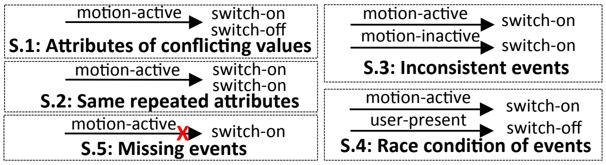


Figure 8: Illustration of general properties (S.1-S.5).

General Properties. General properties are constraints on state models that are independent of an app's semantics–intuitively, these are states and transitions that should never occur regardless of the app domain. We develop the properties based on the constraints on states and state transitions. To illustrate, property S.1 states that a handler must not change an attribute to conflicting values on the same control-flow path, e.g., the motion-active handler must not turn on and turn off a switch in the same branch of the handler. More subtly, property S.4 states that two or more non-complementary handlers must not change an attribute to conflicting values, e.g., a user-present handler turns on the switch while a timer turns off the switch–leading to a potential race condition.

App-specific Properties. App-specific properties are developed according to use cases of one or more devices–here we take a device-centric approach. For instance, P.1 says that the door must always be locked when the user is not at home (thus involving the smart door and presence detector). Similarly, P.30, states that the water valve must be shut off when there is a water leak (thus involving the water valve and moisture sensor). We evaluated all apps using this approach, but defer discussion to the extended paper. We check the app against a property if all of the devices in the property are included in the app.

4.4 Validating Properties

Validation begins by defining a temporal formula for each property to be verified. Thereafter, Soteria uses a general purpose model checker to validate the property with respect to the generated model of the target app (see next section for details). What the user does with a discovered violation is outside the scope of Soteria. However, in most cases, we expect that the results will be recorded

ID	Property Description
P.1	The door must always be locked when the user is not home.
P.10	The alarm must always go off when there is smoke.
P.12	The light must be off when the user is not home.
P.13	The devices (e.g., coffee machine, crock-pot) must always be on at the time set by the user.
P.14	The refrigerator and security system must always be on.
P.17	The AC and heater must not be on at the same time.
P.22	The battery of devices must not be below a specified threshold.
P.28	The sound system must not play music during the sleeping mode.
P.29	The flood sensor must always notify the user when there is water.
P.30	The water valve must be closed if a leak is detected.

Table 1: Examples of application-specific properties. A complete list of properties is available in the extended paper [9].

Algorithm 2: Creating the union of apps' state models

```
   Input   : $G = \{G_i\}_{i=1}^{n}$: State models of $n$ apps
   Output : $G'$ is the union of $\{G_i\}_{i=1}^{n}$
   /* Initialize G' */
1  states($G'$) ← {$v$ | $v$ is a tuple of attribute values in G}
   /* Construct union of apps' state models */
2  for $i \in (1{:}\ n)$ do
3      forall states $v \in G_i$ do
4          forall transitions $e = v \xrightarrow{l} u \in G_i$ do
5              $V'$ is a subset of states in $G'$ that contain $v$
6              $U'$ is a subset of states in $G'$ that contain $u$
7              forall $v' \in V'$ and $u' \in U'$ do
8                  add $e' = v' \xrightarrow{l} u'$ to $G'$ and label the edge with $i$
9              end
10         end
11     end
12 end
```

and the code hand-investigated to determine the cause(s) of the violation. If the violation is not acceptable for the domain or environment, the app can be rejected (from the market) or modified (by the developer) as needs dictate.

Validation of properties in multi-app environments is more challenging. Apps often interact through a common device or some common abstract event (such as the home or away modes). For illustration, consider two apps (App1 and App2) co-resident with the Smoke-Alarm and Thermostat-Energy-Control apps in a multi-device environment. App1 changes the mode from away to home when the light switch is turned on, and App2 turns off a light switch when the smoke is detected, as follows:

Smoke-Alarm: switch-off $\xrightarrow{\text{smoke-detected}}$ switch-on
App1: away-mode $\xrightarrow{\text{switch-on}}$ home-mode
Thermostat-Energy-Control: door-unlocked $\xrightarrow{\text{home-mode}}$ door-locked
App2: switch-on $\xrightarrow{\text{smoke-detected}}$ switch-off

The Smoke-Alarm app interacts with App1 through the switch, and interacts with App2 through the smoke detector and switch. The Thermostat-Energy-Control app interacts with App2 through the mode-change event.

To check general and app-specific properties in the setting of multiple apps, Soteria builds a state model that is the union of the apps' state models. The resulting state model G' represents the complete behavior when running the multiple apps together. The union algorithm is presented in Algorithm 2. Soteria first creates an empty-transition state model G' whose states are the Cartesian product of the states in the input apps (line 1); note that since the input apps' states encode device attributes, the Cartesian product should remove attributes of duplicate devices (i.e., those devices that appear in multiple apps). For instance, if we consider Smoke-Alarm and App1, G' should have four states, and each state encodes a pair of switch and mode attributes. The algorithm then iterates through all apps' transitions and adds appropriate transitions to the union model G'. Soteria's union algorithm is a modification of the multiple-graph union algorithm of igraph library [22], based on a set of constraints on transitions and states. It has a complexity of $O(|V|+|E|)$, $|V|$ and $|E|$ is the number of vertices and edges in G'.

With the union state model created, Soteria then performs model checking on the union model concerning properties we discussed earlier. As an example, Soteria reports that, when Smoke-Alarm and App2 are used together, there is a property violation of S.1: the smoke-detected event would make the Smoke-Alarm app turn on the switch, while it would also make App2 to turn off the switch. As another example, when Smoke-Alarm, App1 and Thermostat-Energy-Control are used together, there is a misuse case that violates property P.3: the door would be locked when there is smoke at home. The property violation is demonstrated as follows:

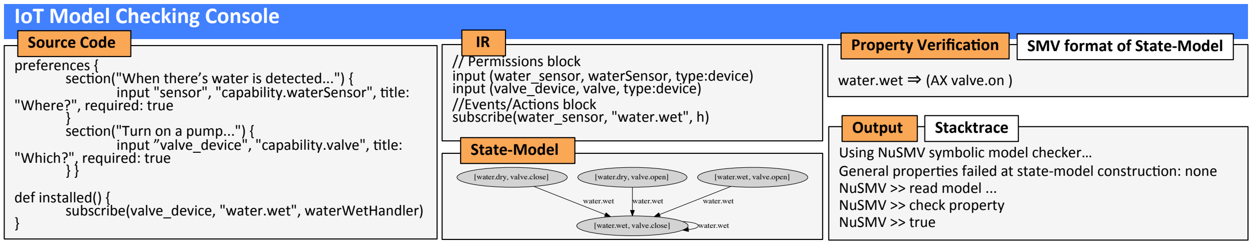
switch-off $\xrightarrow{\textbf{smoke-detected}}$ **switch-on** $\xrightarrow{\textbf{switch-on}}$ **home-mode** $\xrightarrow{\textbf{home-mode}}$ **door-locked**

P.3 is violated because switch-on attribute in the Smoke-Alarm app is used by App1, which changes the mode from away to home. The mode change then triggers locking the door in Thermostat-Energy-Control.

5 Implementation

IR and State Model Construction. Constructing an IR from the source code requires, among other things, the building of the app's ICFG. Here the Soteria IR-building algorithm directly works on the Abstract Syntax Tree (AST) representation of Groovy source code. The Groovy compiler supports customizing the compilation via compiler hooks, through which one can insert extra passes into the compiler (similar to the modular design of the LLVM compiler [27]). Soteria visits AST nodes at the compiler's semantic analysis phase where the Groovy compiler performs consistency and validity checks on the AST. Our implementation uses an ASTTransformation to hook into the compiler, GroovyClassVisitor to extract the entry points and the structure of the analyzed app, and GroovyCodeVisitor to extract method calls and expressions inside AST nodes. Here we AST visitors to analyze expressions and statements to construct the IR and model.

Soteria uses AST visitors for state model construction as well. We extend the ASTBrowser class implemented in the Groovy Swing console, which allows users to enter and run Groovy scripts [19]. The implementation hooks into the IR of an app in the console and dumps information to the TreeNodeMaker class; the information includes an AST node's children, parent, and all properties built during compilation. This includes the resolved classes,



```
preferences {
          section("When there's water is detected...") {
                    input "sensor", "capability.waterSensor", title:
"Where?", required: true
          }
          section("Turn on a pump...") {
                    input "valve_device", "capability.valve", title:
"Which?", required: true
          } }

def installed() {
          subscribe(valve_device, "water.wet", waterWetHandler)
}
```

```
// Permissions block
input (water_sensor, waterSensor, type:device)
input (valve_device, valve, type:device)
//Events/Actions block
subscribe(water_sensor, "water.wet", h)
```

```
water.wet ⇒ (AX valve.on )
```

```
Using NuSMV symbolic model checker...
General properties failed at state-model construction: none
NuSMV >> read model ...
NuSMV >> check property
NuSMV >> true
```

Figure 9: Our SOTERIA framework designed for IoT apps. The left region is the analysis frame; the middle region contains the IR and visual representation of the state model of an example IoT app, and the right region shows the output for a property violation.

static imports, the scope of variables, method calls, interfaces accessed in an app. We then use Groovy visitors to traverse the IR's ICFG and extract the state model.

Model Checking with NuSMV. We translate the state model of an IoT app into a Kripke structure [12]. A Kripke structure is an equivalent temporal structure of a state model and increases readability. We create a visual representation of a state model using open-source graph visualization software GraphViz [14]. We use the open-source symbolic model checker NuSMV [10] for its reliability and maturity. We express properties with temporal logic formulas [11]. NuSMV either confirms a property holds or presents a counter-example showing why the property is false. To address state explosion in apps that control a large number of devices or that have complex control logic, we use NuSMV options that combine binary decision diagrams (BDDs)-based model checking with SAT-based model checking [6]. This was successfully applied to verify models having more than 10^{20} states and hundreds of state variables [7].

Output of SOTERIA. Fig. 9 presents SOTERIA's analysis result on a sample app. It builds the app IR, extracts the state model, and displays a visual representation of the state model. For each property, SOTERIA either shows the property holds or presents a counter-example.

6 Evaluation

As a means of evaluating the SOTERIA framework, we performed an analysis on two large-scale data-sets–one market based and one synthetic. In these studies, we sought to validate the correctness, completeness, and performance of property analysis on the target datasets. We performed our experiments on a laptop computer with a 2.6GHz 2-core Intel i5 processor and 8GB RAM, using Oracle's Java Runtime version 1.8 (64 bit) in its default settings. We use NuSMV 2.6.0 for model checking and Graphviz 2.36 for visualization of a state model.

Datasets. For the market dataset, we obtained 35 official (vetted) apps (O1-O35) from the SmartThings GitHub repository [43] and 30 community-contributed third-party (non-vetted) apps (TP1-TP30) from the official SmartThings community forum [41] in late 2017 (see Table 2). The 65 apps were selected to include various devices and functionality that encompass diverse real-life use-cases.

	Nr.	Unique Devices	Avg/Max States‡	Avg/Max LOC	Func.†
Official	35	14	36/180	220/2633	All
Third-party	30	18	32/96	246/1360	All

‡ This is after applying SOTERIA's state-reduction algorithms.

† The apps cover all spectrum of functionality, including security and safety, green living, convenience, home automation, and personal care. We determined an app's functionality by checking definition blocks in its source code.

Table 2: Description of analyzed official and third-party apps.

For the synthetic dataset, we introduce MALIOT [23], an open source repository containing flawed IoT apps. Inspired by other security-relevant app test suites [4, 15, 28], MALIOT includes 17 hand-crafted flawed SmartThings apps (App1-App17) containing property violations in an individual app and multi-app environments. 14 apps have a single property violation, and three have multiple property violations, with a total of 20 property violations. The apps include various devices covering diverse real-life use-cases. The accurate identification of property violations requires program analysis including multiple entry points, numerical-valued device attributes, and transitions guarded by predicates. Each app in MALIOT also comes with ground truth of what properties are violated; this is provided in a comment block in the app's source code.

6.1 Market App Evaluation

We first report results of the verification of general (S.1-S.5) and app-specific (P.1-P.30) properties. The properties are checked for each app and collections of apps working in concert. SOTERIA flagged that nine individual apps and three multi-app groups violate at least one property. We manually checked the property violations and verified that all reported ones are true positives. The manual checking process was straightforward to perform since SmartThings apps are relatively small.

Individual App Analysis. Table 3 the results of our analysis on single apps. SOTERIA flagged one third-party app violating multiple properties, eight third-party apps violating a single property. None of the official apps were flagged as violating properties; we believe this is because of the strict manual vetting enforced on official apps, which takes a couple of months [40]. For third-party apps, we manually verified that all reported property violations are indeed problems with the implementation. For exam-

ID	Violation Description	Violated Pr.
TP1	The music player is turned on when user is not at home.	P.13
TP2	The switch turns on and blinks lights when no user is present.	P.12
TP3	The location is changed to the different modes when the switch is turned off and when the motion is inactive.	S.4
TP4	The flood sensor sounds alarm when there is no water.	P.29
TP5	The music player turns on when the user is sleeping.	P.28
TP6	The lights turn on and turn off when nobody is at home.	P.13 and S.1
TP7	The lights turn on and turn off when the icon of the app is tapped.	S.1
TP8	The door is unlocked on sunrise and locked on sunset.	P.1
TP9	The door is locked multiple times after it is closed.	S.2

Table 3: SOTERIA's results on individual apps.

ple, a property violation happens in an app (TP6) that turns off and on a light switch when there is nobody at home; another app (TP9) unlocks the door at sunset and locks the door at sunrise–and unintended action.

To assess whether the property violations are real bugs in analyzed apps, we opened a thread in official SmartThings community forum and asked users whether the functionality of the apps confirms their expectations [42]. We got eight answers from the users that are smart home enthusiasts. These apps may have subtle and surprising uses under the right conditions: a user for TP4, said that he used his flood sensor to let him know when there is no water so that he can add water to the trees during Christmas; another user stated that TP6 might simulate occupancy of his home at night by randomly turning on/off lights when nobody is home. To guard against malicious code, those users stated that they attempted to read and understand the source code of the apps before they installed them. However, since regular users cannot be expected to read and check the source code of apps manually, SOTERIA addresses this problem by analyzing apps and presenting their potential property violations to users, which allows them to determine whether a violation is actually harmful.

Multi-App Analysis. We found that multiple apps working in concert can lead to unsafe and undesired device states. SOTERIA flagged three group of apps violating multiple properties. We examined 28 groups and found three groups that have 17 apps violate 11 properties. Table 4 shows the app groups, events, and device attributes that constitute violations, and violated properties. In the following discussion, we will use app group IDs (G.1-G.3) in Table 4. Each group includes a set of apps that a user may install together and authorize to use the same devices.

In G.1, O3 and O4 violate S.1 by setting the switch attribute to conflicting values when the contact sensor is open; there is a similar violation between O4, O8 and TP12 when the contact sensor is closed. O8 and TP12 violates S.2 by turning on the switch multiple times with the "contact sensor close" event. In addition, O3 and O4 violate S.3 by turning on the switch with complement events of "contact sensor close" and "contact sensor open". In G.2, O9, O16, and TP3 violates S.2 by turning on the switch multiple times with the "motion active" event. Additionally, the interaction between O14, O9, O16 and TP3 violates S.4 by invoking "switch on" and "switch off" actions with different device events ("contact sensor open" and "motion active"). There is a similar violation between O14 and TP2 ("contact sensor open" and "app touch"). These events may occur at the same time, which leads to a race condition. In G.3, similar to the other groups, S.1 and S.2 are violated. In addition, multiple app-specific properties are violated. O7 and TP3 change the location mode when the switch is turned off and also when motion is inactive. O30 and TP21 turn off the switch of a set of devices including a security system, smoke detector, and heater when the location is changed; O31 and TP22 turns on devices such as TV, coffee machine, A/C, and heater when the location is changed; both cases violate multiple properties (P.12, P.13, P.14 and P.17) and cause security and safety risks for users. Lastly, O12 and TP19 sets the thermostat to user settings when the switched is turned off and when the motion is inactive. These result in an unauthorized control of thermostat heating and cooling temperature values.

Gr. ID	App ID	Events/Actions	Violated Pr.
G.1	O3	$\xrightarrow{\text{contact sensor open}}$switch on	S.1, S.2, S.3
	O4	$\xrightarrow{\text{contact sensor open}}$switch off	
		$\xrightarrow{\text{contact sensor close}}$switch on	
	O8, TP12	$\xrightarrow{\text{contact sensor close}}$switch off	
G.2	O14	$\xrightarrow{\text{contact sensor open}}$switch off	S.2, S.4
	O9, O16, TP3	$\xrightarrow{\text{motion active}}$switch on	
	TP2	$\xrightarrow{\text{app touch}}$switch on	
G.3	O7, TP3	$\xrightarrow{\text{switch off}}$change location mode	P.12, P.13, P.14, P.17, S.1, S.2
		$\xrightarrow{\text{motion inactive}}$ change location mode	
	O30, TP21	$\xrightarrow{\text{location mode change}}$switch off	
	O31, TP22	$\xrightarrow{\text{location mode change}}$switch on	
	O12, TP19	$\xrightarrow{\text{location mode change}}$set thermostat heating	
		$\xrightarrow{\text{location mode change}}$set thermostat cooling	

Table 4: SOTERIA's results in multi-app environments.

6.2 MALIOT Evaluation

Our analysis of SOTERIA on MALIOT showed that it correctly identified the 17 of the 20 unique property violations in the 17 apps. SOTERIA produces a false warning for an app that uses call by reflection (App5). This app invokes a method via a string. It over-approximates the call graph by allowing the method invocation to target all methods in the app. Since one of the methods turns off the alarm when there is smoke, SOTERIA reports a violation. However, it turns out that the reflective call in this app would not call the property-violating method. Note this pattern did not appear in the 65 real IoT apps we discussed earlier. Additionally, SOTERIA did not report a violation for an app that leaks sensitive data (App10) and for an app that implements dynamic device permissions (App11) as they are outside the scope of SOTERIA analysis.

6.3 MicroBenchmarks

State Reduction Efficacy. Earlier we presented algorithms for performing property abstraction on numerical-valued device attributes. To evaluate its impact, we measured the number of states before and after the application of these algorithms, and the results are presented on the top of Fig. 10. We note that SOTERIA performs state reduction only for apps with devices that have numerical-valued

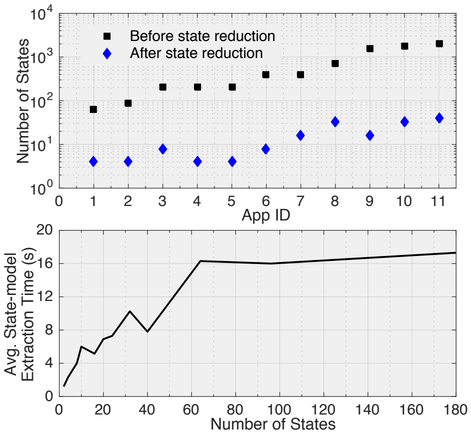


Figure 10: Soteria's state reduction efficacy (Top). Soteria's state model extraction overhead (Bottom).

attributes; examples include thermostats, batteries, and power meters. Among the devices we examine, there are ten such devices in analyzed apps, and 14 apps grant access to these devices, and the states of three apps have the same number before reduction and reduced to the same number. The figure shows that Soteria's state reduction often results in order of magnitude less number of states. **State Model Extraction Overhead.** We ran Soteria with apps that have varying numbers of states and recorded the state-model generation time; the result is shown on the bottom of Fig. 10. The time includes the time for IR extraction, generating a graphical representation of the model, obtaining the SMV code of a state model, and logging (required for general properties). The average run-time for an app with 180 states was 17.3±2 secs. We note that the total time depends on the time taken by the algorithms we have developed for state reduction. For instance, an app having 32 states took more time than an app having 40 states due to many branches used in the 32-state app. Note that overheads can be mitigated by eliminating non-essential processing and other optimization.

We also measured the time for constructing a state model in multi-app environments. The state model of multiple apps requires extraction of each app's state model. Soteria's graph-union algorithm then finds 30 interacting apps (which have on average 64 states and six state attributes) and 4±2.1 seconds for the union algorithm. **Property Verification Overhead.** We evaluated the verification time of a property on state models. The verification of a property took on the order of milliseconds to perform since the SmartThings apps have comparatively smaller state models than the large-scale ones found in other domains such as operating system kernels.

7 Limitations and Discussion

A limitation of Soteria is the treatment of call by reflection. As discussed in Sec. 4.2.3, Soteria constructs an imprecise call graph that allows a reflective call to target any method. This increases the size of state models and may lead to false positives during property checking. We plan to explore string analysis to statically identify possible values of strings and refine the target sets of method calls by reflection. Another limitation of Soteria is dynamic device permissions and app configurations. These may yield property violations because of the erroneous device and input configurations by users at install time. For instance, if a user enters an incorrect time value, the door may be left unlocked in the middle of the night.

Soteria's implementation and evaluation are based on the SmartThings programming platform designed for home automation. There are other IoT domains suitable for applying model checking for finding property violations, such as FarmBeats for agriculture [46], HealthSaaS for healthcare [20], and KaaIoT for the automobile industry [26]. We plan to extend our Soteria to these platforms by applying the IR-based analysis, as well as engage in large-scale analyses of IoT markets and industries.

8 Conclusions

We presented Soteria[4], a novel system that extracts state models from IoT code suitable for finding the security, safety, and functional errors. We evaluated Soteria in two studies; a study of apps on the SmartThings market, and a study on our novel MalIoT app corpus. These studies demonstrated that our approach can efficiently identify property violations and that many apps violate properties when used in isolation and when used together in multi-app environments. In future work, we will extend the kinds of analysis and provide a suite of tools for developers and researchers to evaluate implementations and study the complex interactions between users and IoT environments devices that they use to enhance their lives.

9 Acknowledgments

The authors thank Ashutosh Pattnaik and Prasanna Rengasamy for helpful discussions about this work. We also thank Megan McDaniel for taking care of our diet before the paper deadline. Research was supported in part by the Army Research Laboratory, under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA) and the National Science Foundation Grant No. CNS-1564105. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

[4] The development of Soteria suite and its subsequent evaluation of IoT apps was a highly complex endeavor. An extended version of this paper is available with substantially more description, detail, and commentary, as well as 1) Groovy source code and IR of three example apps, 2) detailed description of general and application-specific properties, 3) advanced SmartThings idiosyncrasies for state-model extraction, and 4) description of the MalIoT apps and their property violations [9].

References

[1] Apple Home Kit. https://www.apple.com/ios/home/. [Online; accessed 29-April-2018].

[2] Apple Home Kit Security and Privacy on IOS. https://www.apple.com/business/docs/iOS_Security_Guide.pdf. [Online; accessed 29-April-2018].

[3] Apple HomeKit App Submission Guideline. https://developer.apple.com/app-store/review/guidelines/#homekit. [Online; accessed 9-April-2018].

[4] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN* (2014).

[5] Beyer, D., Gulwani, S., and Schmidt, D. Combining model checking and data-flow analysis. *Model Checking* (2017).

[6] Biere, A., et al. Symbolic model checking without BDDs. In *Algorithms for the Construction and Analysis of Systems* (1999).

[7] Burch, J., Clarke, E. M., and Long, D. Symbolic model checking with partitioned transition relations. *Research Report, CMU Computer Science* (1991).

[8] Celik, Z. B., Babun, L., Sikder, A. K., Aksu, H., Tan, G., McDaniel, P., and Uluagac, A. S. Sensitive information tracking in commodity IoT. In *USENIX Security* (2018).

[9] Celik, Z. B., McDaniel, P., and Tan, G. Soteria: Automated IoT safety and security analysis (Extended Paper). *arXiv:1805.08876* (2018).

[10] Cimatti, A., et al. NuSMV 2: An open source tool for symbolic model checking. In *International Conference on Computer Aided Verification* (2002).

[11] Clarke, E. M., and Emerson, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs* (1981).

[12] Clarke, E. M., Grumberg, O., and Peled, D. *Model checking*. MIT press, 1999.

[13] Das, M., Lerner, S., and Seigle, M. ESP: Path-sensitive program verification in polynomial time. In *ACM Sigplan Notices* (2002).

[14] Ellson, J., et al. Graphviz open source graph drawing tools. In *International Symposium on Graph Drawing* (2001).

[15] Enck, W., et al. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transaction on Computer Systems* (2014).

[16] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *ACM CCS* (2011).

[17] Fernandes, E., et al. FlowFence: Practical data protection for emerging IoT application frameworks. In *USENIX Security* (2016).

[18] Fernandes, E., Jung, J., and Prakash, A. Security analysis of emerging smart home applications. In *Security and Privacy (S&P)* (2016).

[19] Groovy Console - the Groovy Swing console. http://groovy-lang.org/groovyconsole.html. [Online; accessed 20-April-2018].

[20] HealthSaaS: The Internet of Things (IoT) Platform for Healthcare. https://www.healthsaas.net/. [Online; accessed 20-April-2018].

[21] Ho, G., Leung, D., Mishra, P., Hosseini, A., Song, D., and Wagner, D. Smart locks: Lessons for securing commodity internet of things devices. In *AsiaCCS* (2016).

[22] igraph-the network analysis package. http://igraph.org/r/doc/. [Online; accessed 29-April-2018].

[23] IoTBench: A micro-benchmark suite to assess the effectiveness of tools designed for IoT apps. https://github.com/IoTBench. [Online; accessed 29-April-2018].

[24] Jablokow, A. How the IoT helps keep oil and gas pipelines safe. *Product Lifecycle Report* (November 2015).

[25] Jia, Y. J., et al. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *NDSS* (2017).

[26] KaaIoT: IoT automotive. https://www.kaaproject.org/automotive/. [Online; accessed 20-January-2018].

[27] Lattner, C. *LLVM compiler infrastructure project*. The architecture of open source applications, 2012.

[28] McLaughlin, S., and McDaniel, P. SABOT: specification-based payload generation for programmable logic controllers. In *ACM CCS* (2012).

[29] Mead, N. R. How to compare the security quality requirements engineering (square) method with other methods. Tech. rep., CMU Software Engineering Institute, 2007.

[30] Oluwafemi, T., et al. Experimental security analyses of non-networked compact fluorescent lamps: A case study of home automation security. In *LASER* (2013).

[31] OpenHAB App Submission Guideline. https://marketplace.eclipse.org. [Online; accessed 17-April-2018].

[32] OpenHAB: Home Automation. https://www.openhab.org/. [Online; accessed 15-April-2018].

[33] Oracle Software Security Assurance. http://www.oracle.com/security/software-security-assurance.html. [Online; accessed 15-April-2018].

[34] Orcutt, M. Security experts warn congress that the internet of things could kill people. *MIT Technology Review* (2016).

[35] Ronen, E., and Shamir, A. Extended functionality attacks on IoT devices: The case of smart lights. In *Euro S&P* (2016).

[36] Ronen, E., Shamir, A., Weingarten, A.-O., and O'Flynn, C. IoT goes nuclear: Creating a zigbee chain reaction. In *Security and Privacy (S&P)* (2017).

[37] Samsung SmartThings. https://www.smartthings.com/. [Online; accessed 9-April-2018].

[38] Schumacher, M., et al. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, 2013.

[39] Sharir, M., and Pnueli, A. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications* (1981).

[40] SmartThings Code Review Guidelines. http://docs.smartthings.com/en/latest/code-review-guidelines.html. [Online; accessed 29-April-2018].

[41] SmartThings community-created third-party apps. https://community.smartthings.com. [Online; accessed 29-April-2018].

[42] SmartThings Community Forum User Study Post. https://goo.gl/yC1wFf, 2018.

[43] SmartThings Developers. https://github.com/SmartThingsCommunity. [Online; accessed 29-April-2018].

[44] SmartThings Documentation. http://docs.smartthings.com. [Online; accessed 29-January-2018].

[45] Tian, Y., et al. SmartAuth: user-centered authorization for the Internet of Things. In *USENIX Security* (2017).

[46] Vasisht, D., et al. Farmbeats: An IoT platform for data-driven agriculture. In *NSDI* (2017).

[47] Yoshioka, N., Washizaki, H., and Maruyama, K. A survey on security patterns. *Progress in Informatics* (2008).

Scaling Guest OS Critical Sections with *e*CS

Sanidhya Kashyap   Changwoo Min[†]   Taesoo Kim
Georgia Institute of Technology   *Virginia Tech*[†]

Abstract

Multi-core virtual machines (VMs) are now a norm in data center environments. However, one of the well-known problems that VMs suffer from is the vCPU scheduling problem that causes poor scalability behaviors. More specifically, the symptoms of this problem appear as preemption problems in both under- and over-committed scenarios. Although prior research efforts attempted to alleviate these symptoms separately, they fail to address the common root cause of these problems: the missing semantic gap that occurs when a guest OS is preempted while executing its own critical section, thereby leading to degradation of application scalability.

In this work, we strive to address all preemption problems together by bridging the semantic gap between guest OSes and the hypervisor: the hypervisor now knows whether guest OSes are running in critical sections and a guest OS has hypervisor's scheduling context. We annotate all critical sections by using the *lightweight* para-virtualized APIs, so we called *enlightened critical sections (eCS)*, that provide scheduling hints to both the hypervisor and VMs. The hypervisor uses the hint to reschedule a vCPU to fundamentally overcome the *double scheduling* problem for these annotated critical sections and VMs use the hypervisor provided hints to further mitigate the blocked-waiter wake-up problem. Our evaluation results show that *e*CS guarantees the forward progress of a guest OS by 1) decreasing preemption counts by 85–100% while 2) improving the throughput of applications up to 2.5× in an over-committed scenario and 1.6× in an under-committed scenario for various real-world workloads on an 80-core machine.

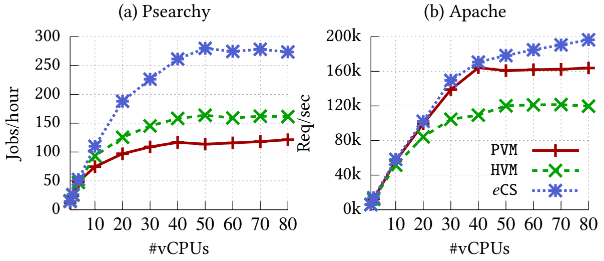


Figure 1: Impact of exposing some of the semantic information from the VM to the hypervisor and vice-versa, which leads to better scalability of Psearchy and Apache web server benchmark, in a scenario in which two VMs are running with the same benchmark. Here, PVM and HVM denote with and without para-virtualization support, while *e*CS represents our approach. Psearchy mostly suffers from LHP and BWW. Similarly, Apache suffers from LHP and ICP.

1 Introduction

Virtualization is now the backbone of every cloud-based organization to run and scale applications horizontally on demand. Recently, this scalability trend is also extending towards vertical scaling [2, 11], *i.e.*, a virtual machine (VM) has up to 128 virtual CPUs (vCPUs) and 3.8 TB of memory to run large in-memory databases [24, 35] and data processing engines [47]. At the same time, cloud providers strive to oversubscribe their resources to improve hardware utilization and reduce energy consumption, without imposing any permissible overhead on the application [38, 46]. However, over subscription requires multiplexing of physical CPUs among VMs to equally distribute physical CPU cycles. Thus, the multiplexing of these VMs introduces the *double scheduling problem* [40]: 1) the guest OS schedules processes on vCPUs and 2) the hypervisor schedules vCPUs on physical CPUs. Some of the prior works address this problem by adopting co-scheduling approaches [17, 41, 45], which can suffer from priority inversion, CPU fragmentation, and may mitigate the double scheduling symptoms [40]. Such symptoms, that have mostly been addressed individually, are lock-holder preemption (LHP) [8, 15, 42, 44], lock-waiter preemption (LWP) [44], and blocked-waiter wakeup (BWW) [5, 39], problems.

The root cause of this double scheduling phenomenon is a semantic gap between a hypervisor and guest OSes, in which the hypervisor is agnostic of not only the scheduling of VMs but also guest OS-specific critical code that deter the scalability of applications. Furthermore, LHP/LWP are not only limited to spinlocks [15, 19, 42], but are also possible in blocking primitives such as mutex and rwsem as well as readers of the rwsem. Moreover, because of their non work-conserving nature, these blocking primitives inherently suffer from the BWW problem (refer Psearchy in Figure 1 (a)). Besides these, none of the prior works have identified the preemption of an interrupt context that happens in interrupt-intensive applications such as Apache web-server (Figure 1 (b)). We define this problem as *interrupt context preemption* (ICP).

Our key observation is that these symptoms occur because 1) the hypervisor is scheduling out a vCPU at a time when the vCPU is executing a critical code, and 2) a vCPU, waiting to acquire a lock, is either uncooperative or sleeping [16], leading to LWP and BWW issues. Thus, we propose an alternative perspective, *i.e.*, instead of devising a solution for each symptom, we use four key ideas that allows a VM to hint the hypervisor for making an effective scheduling decision to allow its forward progress. First, we consider all of the locks and interrupt contexts as critical components. Second, we devise a set of para-virtualized APIs that annotate these critical components as *enlightened critical sections* (*e*CS). These APIs are lightweight in nature and notify a hypervisor from the VM and vice-versa with memory operations via shared memory, while avoiding the overhead of hypercall and interrupt injection. Third, the hypervisor now can figure out whether a vCPU is executing an *e*CS and can reschedule it. We empirically found that an extra schedule (one millisecond [27]) is sufficient as it decreases preemptions by 85–100%; and these critical sections are shorter (in μs [42]) than one schedule. However, by rescheduling a vCPU, we introduce unfairness in the system. We tackle this issue with the OS's fair scheduling policy [27], which compensates for that additional schedule by allowing other tasks to run for extra time, thereby maintaining the eventual fairness in the system. Lastly, we leverage our APIs to design a virtualized schedule-aware spinning strategy (*e*SchdSpin) that enables lock waiters to be work conserving as well as cooperative inside a VM. That is, a vCPU now cooperatively spins for the lock, if a physical CPU is under-committed, else it yields the vCPU.

Thus, our approach improves the scalability of real-world applications by 1.2–1.6× in an under-committed case. Moreover, our *e*CS annotation, combined with *e*SchdSpin, avoids preemption by 85–100% while improving the scalability of applications by 1.4–2.5× in an over-committed scenario on an 80-core machine.

In summary, we make the following contributions:

- We identify similarities among various subproblems that stem from the double scheduling phenomenon. Moreover, we identify three new problems: 1) LHP in blocking locks, 2) readers preemption (RP) in read-write locks and semaphores, and 3) vCPU preemption while processing an interrupt context (ICP).
- We address these subproblems with *e*CS, which we annotate with six new APIs that bridge the semantic gap between a hypervisor and a VM, and even among vCPUs inside a VM.
- Our annotation approach, along with *e*SchdSpin, improves the scalability of applications in both under- and over-committed scenarios up to 2.5× with only 0–15% preemptions, while maintaining eventual fairness with merely one extra schedule.

2 Background and Motivation

We first describe the problem of double scheduling and highlight its implications. Later, we summarize the prior attempts to solve this problem, and then motivate our approach.

2.1 Symptoms of Double Scheduling

In a virtualized environment, a hypervisor multiplexes the hardware resources for a VM, such as assigning vCPUs to physical CPUs (pCPUs). In particular, it runs a vCPU to execute by its fair share [27], which is a general policy of commodity OSes such as Linux, and preempts it because of either vCPUs of other VM or of the intermittent processes of the OS and bookkeeping tasks of the hypervisor such as I/O threads. Hence, there is a possibility that the hypervisor can preempt a vCPU while executing some critical task inside a VM that leads to an application performance anomaly, which we enumerate below:

Lock holder preemption (LHP) problem occurs when a vCPU holding a lock gets preempted and all waiters waste CPU cycles for the lock. Most of the prior works [8, 14, 42, 44] have focused on non-blocking primitives such as spinlocks.[1] On the other hand, LHP also occurs in blocking primitives such as mutex [28] and rwsem [26, 31], which the prior works have not identified. However, LHP accounts up to 90% preemptions for blocking primitives in some of the memory intensive applications that have short critical sections.

Lock waiter preemption (LWP) problem stems when the very next waiter is preempted just before acquiring the lock, which occurs due to the strict FIFO ordering of spinlocks [14, 42]. Fortunately, this problem has been mostly mitigated in existing spinlock design [19, 20], as the current implementation allows waiters to steal the lock before joining the waiter queue. We do not see such a problem in blocking primitives because the current implementation is based on the test-and-set (TAS) lock—an unfair lock, which inherently mitigates LWP.

Blocked-waiter wakeup (BWW) problem occurs mostly for blocking primitives in which the latency to wake up a waiter to pass the lock is quite high. This issue severely degrades the throughput of applications running on a high core count [16], even in a native environment. Moreover, it is evident in both under- and over-committed VM scenarios. For example, the BWW problem degrades the application scalability up to 1.6× (refer Figure 6).

Readers preemption (RP) problem is a new class of problem that occurs when a vCPU holding a read lock among multiple readers gets preempted. This problem impedes the forward progress of a VM and also increases the latency of the write lock. For instance, various memory-intensive workloads have sub-optimal throughput as RP accounts to at most 20% of preemptions. We observe this issue in various read-dominated memory-intensive workloads in which the readers are scheduled out.

RCU reader preemption (RRP) problem is a type of RP

[1] Non-blocking locks, both holders and waiters, do not schedule out. However, the para-virtualized interface converts spinlocks to blocking locks (only waiters) with hypercalls [6, 20] to overcome LHP/LWP issues.

problem that occurs when an RCU reader is preempted, while holding the RCU read lock [33]. Because of RRP, the guest OS suffers from an increased quiescence period. This issue can increase the memory footprint of the application, and is responsible for 5% of preemptions.

Interrupt context preemption (ICP) problem happens when a vCPU that is executing an interrupt context gets preempted. In particular, this problem is different from prior works that focus on interrupt delivery [12, 43] rather than interrupt handling. This issue occurs in cases such as TLB shootdowns, function call interrupts, rescheduling interrupts, IRQ work interrupts, etc. in every commodity OS. For example, we found that Apache web server, an interrupt-intensive workload, suffers from the ICP problem as it accounts to almost 18% of preemptions for evaluated workloads (refer Figure 3).

2.2 Prior Approaches

Some of the prior studies mitigate LHP and LWP problems by relaxed co-scheduling [45], balancing vCPUs to physical CPUs [41] with IPIs as a heuristic [17], or using hardware features [34]. Meanwhile, others designed a para-virtualized interface [8, 14, 42, 44] to only tackle the LHP and LWP problem for spinlocks. Besides these, one radical design focused on scheduling VM's processes than vCPUs by hot plugging vCPUs on the basis of load on the VM [3, 40]. Unfortunately, all of these prior works address the double scheduling problem either *partially* that misses other preemption problems, or take a radical path that is not only difficult to adopt in practice but can have significant overhead, in terms of scaling for machines with almost 100 physical cores. Because their approach involves 1) the detection of response to the double scheduling in the form of hypercalls and interrupt injection [3], and 2) explicit task migration from idle vCPUs to active vCPUs. On the contrary, our approach does simple memory operations and exploits the vCPU scheduling boundary to notify the hypervisor for scheduling decisions without any explicit task and vCPU migration: a lightweight approach even at high core count.

2.3 The Case for *An Extra Schedule*

As mentioned before, OS critical sections are the ones that define the forward progress of an application for which the OS is responsible. For instance, let us take an example of two threads competing to acquire a lock to update contents of a file. If the lock holder, which is updating the file, is preempted, the other waiter will waste CPU cycles. There are several critical operations that affect the application scalability [16, 25], and OS performs such operations either by acquiring a lock or executing an interrupt context (I/O processing, TLB shootdowns, etc.). In particular, a delay in processing of these critical sections can result in a severe performance anomaly such as a convoy effect [14, 16], or decreased network throughput for applications such as web servers (refer Figure 1). Hence, unlike prior approaches, we propose a simple and an intuitive approach, *i.e.*, now a VM hints the hypervisor about a critical section that enables the hypervisor to let a vCPU execute for a pre-defined time slot (schedule). This extra schedule is sufficient to complete a critical section because 1) most critical sections are very fine-grained, and have a time granularity of several microseconds [42], while 2) the granularity of a single schedule is in the order of milliseconds, which is sufficient enough to complete a critical section. For instance, an extra schedule decreases the preemption count by 85–100% (Figure 3). This approach is not only practical but also critical to apply on machines with large core count. However, the extra schedule introduces *unfairness* in the system, which we address by designing a simple, *zero-overhead schedule penalization algorithm* that tries to maintain the *eventual fairness* in the system by leveraging the CFS [27] that tries to maintain fairness in the system.

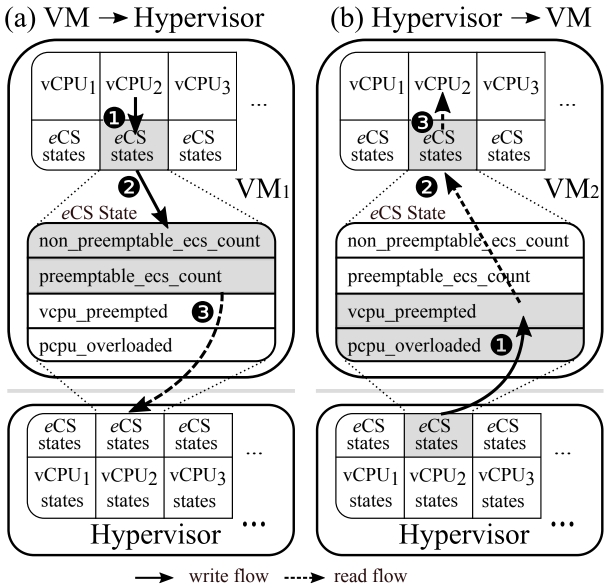


Figure 2: Overview of the information flow between a VM and a hypervisor. Each vCPU has a per-CPU state that is shared with the hypervisor, denoted as *e*CS state. Figure (a) shows how the vCPU$_2$ relays information about an *e*CS to the hypervisor. On entering a critical section or an interrupt context (❶), vCPU$_2$ updates the non_preemptable_ecs_count (❷). After a while, before scheduling out vCPU$_2$, the hypervisor reads its *e*CS state (❸), and allows it run for one more schedule to mitigate any of the double scheduling problems. Figure (b) shows how the hypervisor shares the information whether a vCPU is preempted or a physical CPU is overloaded, at the schedule boundary. For instance, the hypervisor marks vcpu_preempted, while scheduling out a vCPU; or updates pcpu_overloaded flag to one if the number of active tasks on that physical CPU is more than one. Both try to further mitigate LWP and BWW problems.

Hint	Lightweight Para-virtualized API	Description
VM → Hypervisor	`void activate_non_preemptable_ecs(cpu_id)`	Increase the *e*CS count for a `vCPU` with cpu_id by 1 for a non-preemptable task
	`void deactivate_non_preemptable_ecs(cpu_id)`	Decrease the *e*CS count for a `vCPU` with cpu_id by 1 for a non-preemptable task
	`void activate_preemptable_ecs(cpu_id)`	Increase the *e*CS count for a `vCPU` with cpu_id by 1 for a preemptable task
	`void deactivate_preemptable_ecs(cpu_id)`	Decrease the *e*CS count for a `vCPU` with cpu_id by 1 for a preemptable task
Hypervisor → VM	`bool is_vcpu_preempted(cpu_id)`†	Return whether a `vCPU` with cpu_id is preempted by the hypervisor
	`bool is_pcpu_overcommitted(cpu_id)`	Return whether a physical CPU, running a `vCPU` with cpu_id, is over-committed

Table 1: Set of para-virtualized APIs exposed by the hypervisor to a VM for providing hints to the hypervisor to mitigate double scheduling. These APIs provide hints to the hypervisor and VM via shared memory. A `vCPU` relies on the first four APIs to ask for an extra schedule to overcome `LHP`, `LWP`, `RP`, `RRP`, and `ICP`. Meanwhile, a `vCPU` gets hints from the hypervisor by using the last two APIs to mitigate `LWP` and `BWW` problems. The `cpu_id` is the core id that is used by tasks running inside a guest OS.
†Currently, `is_vcpu_preempted()` is already exposed to the VM in Linux.

3 Design

A hypervisor can mitigate various preemption problems, if it is aware of a `vCPU` executing a critical section. We denote such a hypervisor-aware critical section as an *enlightened critical section* (*e*CS), that can be executed for one more schedule. *e*CS is applicable to all synchronization primitives and mechanisms such as `RCU` and interrupt contexts. We now present our lightweight APIs that act as a cross-layer interface for annotating an *e*CS and later focus on our notion of an extra schedule and our approach to maintain eventual fairness in the system.

3.1 Lightweight Para-virtualized APIs

We propose a set of six *lightweight para-virtualized APIs* to bridge the semantic gap that both VM and hypervisor use for conveying information between them. These APIs rely on four variables (refer Figure 2) that are *local* to each `vCPU`. They are exposed via shared memory between the hypervisor and a VM and the notification happens via simple read and write memory operations. A simple memory read is sufficient for the hypervisor to decide on scheduling because 1) it tries to execute each `vCPU` on a separate `pCPU`, 2) and it requires knowing about an *e*CS only at the schedule boundary, thereby removing the cost of polling and other synchronous notifications [3]. To consider an OS critical section as an *e*CS, we mark the start and unmark the end of a critical section, which lets the hypervisor know about an *e*CS. However, a process in an OS can be of two types. First is the non-preemptable process that can never be scheduled out. Such a process is either an interrupt or a kernel thread running after acquiring a spinlock. Another one is the preemptable task such as a user process or a process with blocking lock. Hence, we introduce four APIs (VM → Hypervisor) to separately handle these two types of tasks. The last two APIs (Hypervisor → VM) provide the hypervisor context to the VM, which a lock waiter can use to mitigate the `LWP` problem or yield the `vCPU` to other hypervisor tasks or `vCPU`s in an over-committed scenario. Figure 2 illustrates those four states:

- **`non_preemptable_ecs_count`** maintains the count of active non-preemptable *e*CSs, such as non-blocking locks, `RCU` reader, and interrupt contexts. It is similar to the preemption count of the OS.
- **`preemptable_ecs_count`** is similar to the preemption count variable of the OS, but it only maintains the count of active preemptable *e*CSs, such as blocking primitives, namely, `mutex` and `rwsem`.
- **`vcpu_preempted`** denotes whether a `vCPU` is running. It is useful for handling the `BWW` problem in both under- and over-committed scenarios.
- **`pcpu_overloaded`** denotes whether a physical CPU, executing that particular `vCPU`, is over-committed. Lock waiters can use this information to address the `BWW` problem in an over-committed scenario.

Figure 2 presents two scenarios in which the schedule context information is shared between a `vCPU` and the hypervisor. Figure 2 (a) shows how a `vCPU`, *i.e.*, entering an *e*CS, shares information with the hypervisor. During entry (❶), `vCPU`$_2$ first updates its corresponding state (`non_preemptable_ecs_count` or `preemptable_ecs_count`) (❷) and continues to execute its critical section. Meanwhile, the hypervisor, before scheduling out `vCPU`$_2$, checks `vCPU`$_2$'s *e*CS states (❸) and allows it to run for extra time if certain criteria are fulfilled (§3.2); otherwise, it schedules out `vCPU`$_2$ with other waiting tasks. When `vCPU`$_2$ exits the *e*CS, it decreases the *e*CS state count, denoting the end of critical section. Figure 2 (b) illustrates another scenario that addresses the `BWW` problem. in which the hypervisor updates the *e*CS states: `pcpu_overloaded` and `vcpu_preempted` while scheduling in and out `vCPU`$_2$, respectively, at each schedule boundary (❶). We devise a simple approach—virtualized scheduling-aware spinning (*e*SchdSpin)—that enables efficient scheduling aware waiting for both blocking and non-blocking locks (§4). That is, `vCPU`$_2$ reads both states (❷) and decides whether to keep spinning until the lock is acquired if the `pCPU` is not overloaded (❸), else it yields, which allows the other `vCPU` (in `VM`$_2$) or a hypervisor's task to progress forward by doing some useful task, thereby mitigating the double scheduling problems.

3.2 Eventual Fairness with Selective Scheduling

As mentioned before, the hypervisor relies on its scheduler to figure out whether a `vCPU` is executing an *e*CS. That is, when a `vCPU` with a marked *e*CS is about to be scheduled out, the hypervisor scheduler checks the value of *e*CS count variables (Figure 2). If any of these values are greater than zero, the hypervisor lets the vCPU run for an extra schedule. However, vCPU rescheduling introduces two problems: 1) How does the hypervisor handles a task with *e*CS, which the guest OS can preempt or schedule out? 2) How does it ensure the system fairness?

We handle an *e*CS preemptable task with preemptable_ecs_count counter APIs, which differentiate between a preemptable task and a non-preemptable task. We do so because the guest OS can schedule out a preemptable task. In this case, the hypervisor should avoid rescheduling that vCPU because 1) it will result in false rescheduling, and 2) it can hamper the VM performance. We address this issue inside the guest OS, *i.e.*, before scheduling out an *e*CS-marked task inside a guest OS, we save the value of preemptable_ecs_count to a task-specific structure and reset the counter to zero. Later, when the task is rescheduled again by the guest OS, we restore the preemptable_ecs_count with the saved value from the task-specific structure, thereby mitigating the false scheduling.

With vCPU rescheduling, we introduce unfairness at two levels: 1) An *e*CS marked vCPU will always ask for rescheduling on every schedule boundary.[2] 2) By rescheduling a vCPU, the hypervisor is unfair to other tasks in the system. We resolve the first issue by allowing the hypervisor to reschedule an *e*CS-marked vCPU only once during that schedule boundary as rescheduling extends the boundary. At the end of schedule boundary, the hypervisor schedules other tasks to avoid the starving other tasks or VMs and addresses indefinite rescheduling. In addition, the hypervisor also keeps track of this extra reschedule information and runs other vCPUs for longer duration and inherently balances the running time, an equivalent to vCPU penalization. Thus, our approach selectively reschedules and penalizes a vCPU rather than balancing the extra reschedule information across all cores, which will result in an unnecessary overhead of synchronizing all runtime information of rescheduling. We call our approach as the *local CPU penalization* approach, as we only penalize a vCPU that executed an *e*CS, thereby ensuring *eventual fairness* in the system. Moreover, our local vCPU scheduling is a form of selective-relaxed co-scheduling of vCPUs depending on what kind of tasks are being executed, while without maintaining any synchronization among vCPUs, unlike prior approaches [41, 45].

API	LHP	RP	RRP	ICP	LWP	BWW
activate_non_preemptable_vcs()	✓	✓	✓	✓	-	-
deactivate_non_preemptable_vcs()	✓	✓	✓	✓	-	-
activate_preemptable_vcs()	✓	✓	-	-	-	-
deactivate_preemptable_vcs()	✓	✓	-	-	-	-
is_vcpu_preempted()	-	-	-	-	✓	✓
is_pcpu_overcommitted()	-	-	-	-	-	✓

Table 2: Applicability of our six lightweight para-virtualized APIs that strive to address the symptoms of double scheduling.

Component	Lines of code
*e*CS annotation	60
*e*CS infrastructure	800
Scheduler extension	150
Total	1,010

Table 3: *e*CS requires small modifications to the existing Linux kernel, and the annotation effort is also minimal: 60 LoC changes to support the 10 million LoC Linux kernel that has around 12,000 of lock instances with 85,000 lock invocations.

4 Use Case

The double scheduling phenomenon introduces the semantic gap in three places: 1) from a vCPU to a physical CPU that results in LHP, RP, and ICP problems; 2) from a pCPU to a vCPU; and 3) from one vCPU to another in a VM, both suffer from LWP and BWW problems. Table 2 shows how to use our APIs to address these problems.

LHP, RP, RRP, and ICP problem. To circumvent these problems, we rely on the VM → hypervisor notification because a vCPU running any spinlocks, read-write locks, mutex, rwsem, or an interrupt context is already inside the critical section. Thus, we call activate_*() and deactivate_*() APIs for annotating critical sections. For example, the first two APIs are applicable to spinlocks, read-write locks, RCU, and interrupts, and the next two are for mutex and rwsem. (refer Table 2).

LWP and BWW problem. The LWP problem occurs in the case of FIFO-based locks such as MCS and Ticket locks [23]. However, unfair locks, such as qspinlock [6], mutex [29], and rwsem [37], do not suffer from this problem, and are currently used in Linux. The reason is that they allow other waiters to steal the lock, while suffering from the issue of starvation. On the other hand, all of these locks suffer from the BWW problem because the cost to wake up a sleeping in a virtualized environment varies from 4,000–10,000 cycles. as a wake-up call results in a VMexit, which adds an extra overhead to notify a vCPU to wake up a process. This problem is severe for blocking primitives because they are non-work conserving in nature [16], *i.e.*, the waiters schedule out themselves, even if a single task is present in the run queue of the guest OS. We partially mitigate this issue by allowing the waiters to spin rather than sleep if a single task is present in the run queue of the guest scheduler (SchdSpin). However, this approach is non-cooperative when multiple VMs are running. Thus, to avoid unnecessary spinning of waiters, we rely on our is_pcpu_overcommitted() API that notifies a waiter to only spin if the pCPU is not over-committed. We call this approach the virtualized scheduling-aware spinning approach (*e*SchdSpin).

[2] Such a VM can be either an I/O or an interrupt-intensive VM that spends most of its time in the kernel, or even a compromised VM.

5 Implementation

We realized the idea of *e*CS by implementing it on the Linux kernel version 4.13. Besides annotating various locks and interrupt contexts with *e*CS, we specifically modified the scheduler and the para-virtual interface of the KVM hypervisor. Our changes are portable enough to apply on the Xen hypervisor too. The whole modification consists of 1,010 lines of code (see Table 3).

Lightweight para-virtualized APIs. We share the information between the hypervisor and a VM with a shared memory between them, which is similar to the `kvm_steal_time` [4] implementation. For instance, each VM maintains a per-core *e*CS states, and the hypervisor maintains per-vCPU *e*CS states for each VM.

Scheduler extension. We extend a scheduler-to-task notification mechanism, `preempt_notifier` [18], for identifying an *e*CS-marked vCPU at the schedule boundary. Our extension allows the scheduler to know about the task scheduling requirement and decide scheduling strategy at the schedule boundary. For example, in our case, the extension reads the `non_preemptable_ecs_count` and `preemptable_ecs_count` to decide the scheduling strategy for the vCPU. Besides this, we rely on the notifier's in and out APIs to set the value of `vcpu_preempted` and `pcpu_overloaded` variables.

We implemented our vCPU rescheduling decision in the `schedule_tick` function [36]. The `schedule_tick` function performs two tasks: 1) It does the bookkeeping of the task runtime, which is used for ensuring the fairness in the system. 2) It also is responsible for setting the rescheduling flag (`TIF_NEED_RESCHED`) if there is more than one task on that run queue, which is used by the scheduler to schedule out the task if the reschedule flag is set. We implemented the rescheduling strategy by bypassing the setting up of the reschedule flag in case the `preempt_notifier` check function returned true, meanwhile updating the runtime statistics of the vCPU.

Annotating locks for *e*CS. We mark *e*CS by using the non-preemptable APIs for non-blocking primitives, preemptable ones for `mutex` and `rwsem`. Our annotation comprises only 60 LoC that covers around 12,000 lock instances with 85,000 lock API calls in the Linux kernel that has 10 million LoC for the kernel version 4.13.

6 Evaluation

We evaluate our approaches by answering the following questions:

- What is the overhead of an *e*CS annotation and the scheduler overhead to read the values? (§6.1)
- Does *e*CS helps in an over-committed case? (§6.2)
- How does *e*CS impact the scalability of a VM? (§6.3)
- How do our APIs address the BWW problem? (§6.4)
- Does our schedule penalization approach maintain the eventual fairness of the system? (§6.5)

Experimental setup. We extended VBench [13] for our evaluation. We chose four benchmarks: Apache web server [7], Metis [21], Psearchy from Mosbench, and Pbzip2 [9]. The Apache web server serves a 300 bytes static page for each request that is generated by WRK [10]. Both of them are running inside the VM to remove the network wire overhead and only stress the VM's kernel components. We choose Apache to stress the interrupt handler to emphasize the importance of *e*CS for an interrupt context. Metis is a map-reduce library for a single multi-core server that mostly stresses the memory allocator (spinlock) and the page-fault handler (`rwsem`) of the OS. Similar to Metis, Psearchy is an in-memory parallel search and indexer that stresses the writer side of the `rwsem` design. In addition, we also choose Pbzip2—a parallel compression and decompression program—because we wanted to use a minimally kernel-intensive application. Moreover, none of these workloads suffer from performance degradation from any known user space bottleneck in a non-virtualized environment. We use memory-based file system, `tmpfs`, to isolate the effect of I/O. We further pin the cores to circumvent vCPU migration at the hypervisor level to remove the jitter from our evaluation.

We evaluate our *e*CS approach against the following configurations: 1) PVM is a para-virtualized VM that includes unfair `qspinlock` implementation, which mitigates LWP and BWW issues, and it is the default configuration since Linux v4.5. 2) HVM is the one without paravirtualization support and also includes unfair `qspinlock` implementation. Both PVM and HVM are not *e*CS annotated. Note that we could not compare other prior works because they are not open sourced [3, 45] and are very specific to the Xen hypervisor [42]. We evaluate these configuration on an eight socket, 80-core machine with Intel E7-8870 processors. Another point is that the current version of KVM partially addresses the BWW problem that can occur from the user space [22].

6.1 Overhead of *e*CS

We evaluate the cost of our lightweight para-virtualized APIs on various blocking and non-blocking locks, and RCU. Table 4 enumerates the overhead of the sole API cost including the cost of executing a critical section with a simple microbenchmark that executes an empty critical section to quantify the impact of *e*CS API on these primitives in both lowest (1 core) and highest contention (80 core) scenarios. *1 core* denotes that a thread is trying to acquire a critical section, whereas *80 core* denotes that 80 threads are competing. We observe that *e*CS adds an overhead of almost 0.9–18.4 ns in low contention, whereas negligible overhead in high contention scenario, except RCU. For RCU, the empty critical section

	Time (ns)			
Critical sections	**1 core**		**80 core**	
	W/o *e*CS	W/ *e*CS	W/o *e*CS	W/ *e*CS
API cost	–	16.4	–	16.4
spinlock	31.2	44.8	4,782.3	4,772.9
rwlock (read)	32.0	38.8	2,418.2	2,519.4
rwlock (write)	27.4	45.8	4,363.3	4,784.5
mutex	33.5	34.4	49,116.4	48,125.7
rwsem (read)	35.6	36.6	2,588.8	2,737.0
rwsem (write)	33.3	38.1	7,055.7	7,150.1
RCU	9.8	19.7	9.8	19.8

Table 4: Cost of using our lightweight para-virtualized APIs with various synchronization primitives and mechanism. *1 core* and *80 core* denote the time (in ns) to execute an empty critical section with one and 80 threads, respectively. Although, our approach slightly adds an overhead on a single core count, there is no performance degradation for our evaluated workloads.

suffers from almost twice the overhead because both RCU's lock/unlock operations do a single memory update on the preempt_count variable for a preemptable kernel. Even though our APIs add an overhead in the low contended scenario, we do not observe any performance degradation for any of our evaluated workloads.

6.2 Performance in an Over-committed Scenario

We evaluate the performance of the aforementioned workloads in an over-committed scenario by running two VMs in which each vCPU from both VMs share a physical CPU. Figure 3 (i) shows the throughput of these workloads for PVM, HVM, and *e*CS; (ii) shows the number of unavoidable preemptions that we capture while running these workloads when a vCPU is about to be scheduled out for *e*CS; and (iii) represents the percentage of types of observed preemptions, namely, LHP for blocking (B-LHP) and non-blocking (NB-LHP) locks, RP, RRP, ICP problems that we observe for the *e*CS configuration, including both avoided and unavoided preemptions.

Apache. *e*CS outperforms both PVM and HVM by 1.2× and 1.6×, respectively (refer (t:a) in Figure 3). Moreover, our approach reduces the number of possible preemptions by 85.8–100% (refer (n:a)) because of our rescheduling approach. We cannot completely avoid all preemptions because of our schedule penalization approach, as some of the preemptions occur consecutively. Even though *e*CS adds overhead, especially to RCU, it still does not degrade the scalability for four reasons: 1) We address the BWW problem, which allows for more opportunities to acquire the lock on time; 2) both hypervisor → VM APIs allow cooperative co-scheduling of the VMs; 3) our extra schedule approach avoids 85.8–100% of captured preemptions with the help of our VM → hypervisor APIs; and 4) the APIs overhead partially mitigates the highly contended system at higher core count by acting as a back-off mechanism. Another interesting observation is that we observe almost every type of preemption (refer Figure 3 (p:a)) because of serving the static pages, which involves blocking locks for the socket connection and softirq and spinlocks use for the interrupts processing. In particular, the number of preemptions is dominated by LHP for non-blocking and blocking locks, followed by ICP and then RP. We believe that the ICP problem will further exacerbate with optimized interrupt delivery mechanisms [12, 43]. PVM is 1.36× faster than HVM at 80 cores because of the support of para-virtualized spinlock (qspinlock [20]) as well as the asynchronous page fault mechanism that decreases the contention [30].

The major bottleneck for this workload is the interrupt injection, which can be mitigated by proposed optimized methods [12, 43]. In addition, Figure 4 (b) presents the latency CDF for the Apache workload at 80 cores in both under- and over-compression case. We observe that *e*CS not only maintains almost equivalent latency as that of PVM in an under-committed case, but also decreases in the over-committed case by 10.3–17% and 9.5–27.9% against PVM and HVM, respectively.

Psearchy mostly stresses the writer side of rwsem as it performs 20,000 small and large mmap/munmap operations along with stressing the memory allocator for inode operations, which mostly idles the guest OS because of the non-work conserving blocking locks [16]. Figure 3 (t:b) shows the throughput, in which *e*CS outperforms both PVM and HVM by 2.3× and 1.7×, respectively. The reason is that we 1) partially mitigate the BWW problem with our *e*SchdSpin approach, and 2) decrease the number of preemptions by 95.7–100% with an extra schedule (refer (n:b)). In addition, our *e*SchdSpin approach decreases the idle time from 65.4% to 45.2%, as it allows waiters to spin than schedule out themselves, which severely degrades the scalability in a virtualized environment, as observed for both PVM and HVM. This workload is dominated by mostly blocking and non-blocking locks, as they account to almost 98% preemptions (refer (p:b)). We also observe that HVM outperforms PVM by 1.33× because the asynchronous page fault mechanism introduces more BWW issue as it schedules out a vCPU if the page is not available, which does not happen for HVM.

Metis is a mix of both page fault and mmap operations that stress both the reader and the writer of the rwsem. Hence, it also suffers from the BWW problem, as we observe in Figure 3 (t:c). *e*CS outperforms PVM and HVM by 1.3× at 80 cores because of the reduced BWW problem and decreased preemptions that account to 91.4–99.5% (Figure 3 (n:c)). Note that the reader preemptions are 20%, thereby illustrating that readers preemptions is possible for read-dominated workloads, which has not been observed by any prior works. We do not observe any difference in the throughput of HVM and PVM.

Pbzip2 is an efficient compression/decompression work-

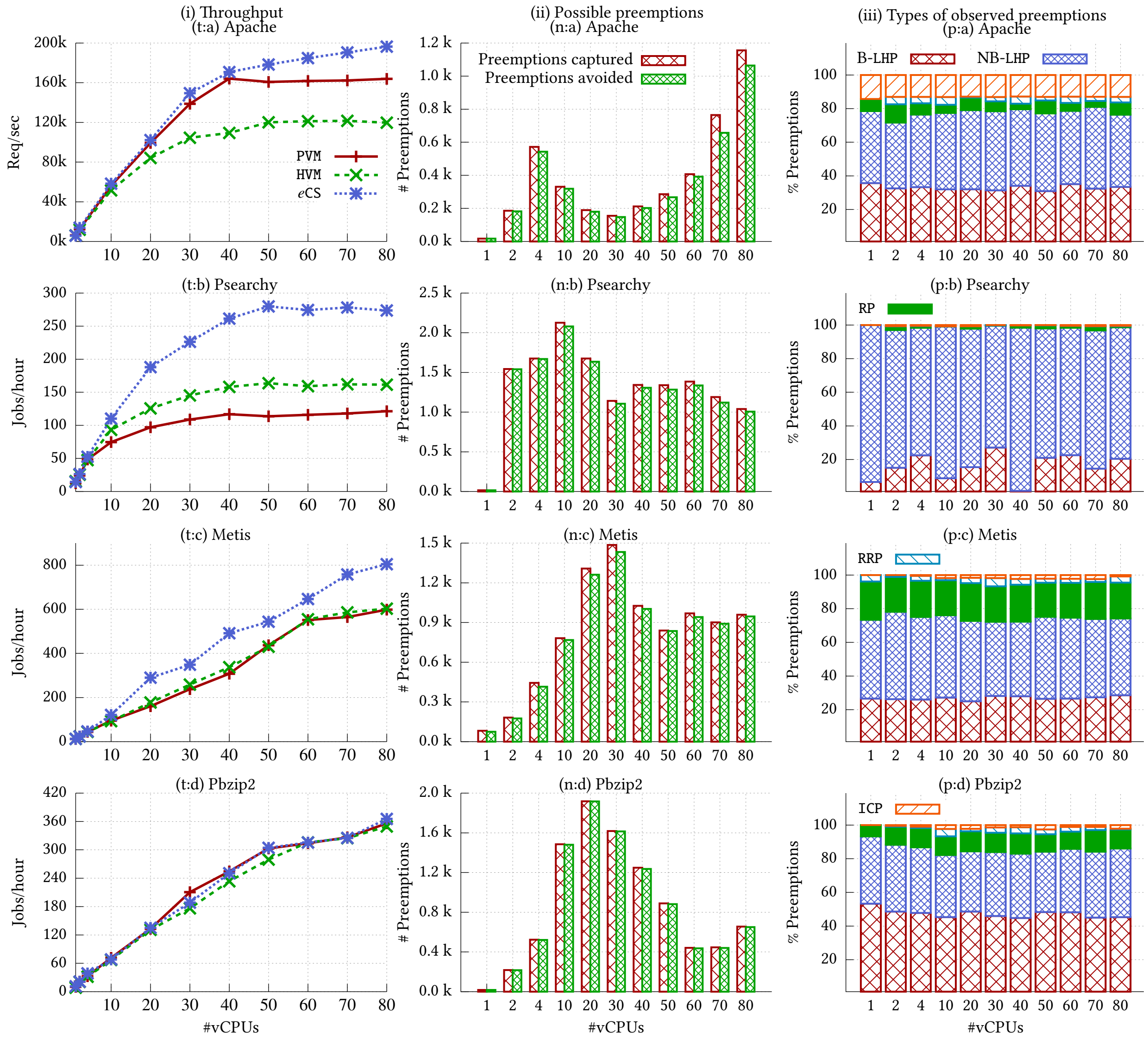


Figure 3: Analysis of real-world workloads in an over-committed scenario, *i.e.*, two instances of VM are executing the same workload. Column (i) represents the scalability of selected workloads in three settings: PVM, HVM, and with *e*CS annotations. Column (ii) represents the number of preemptions caught and prevented by the hypervisor with our APIs. Column (iii) represents the type of preemptions caught by the hypervisor (refer Table 4). By allowing an extra schedule, our approach reduces preemptions by 85–100% and improve scalability of applications by up to 2.5×, while observing almost all types of preemptions for each workload.

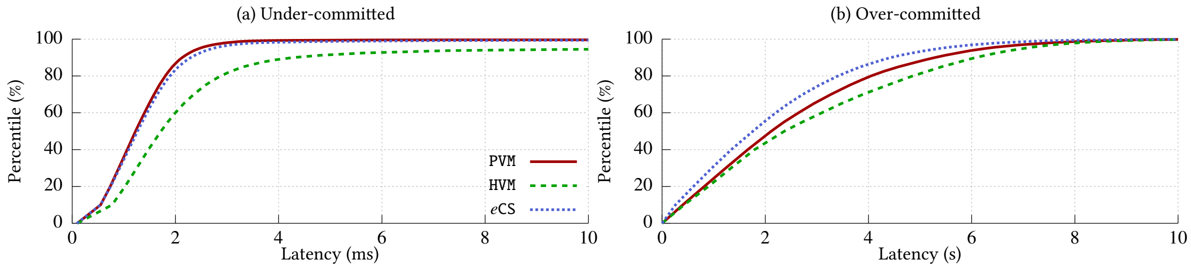


Figure 4: CDF of the latency of requests for the Apache web server workload in both under- and over-committed scenarios at 80 cores. It clearly shows the impact of *e*CS in the over-committed scenario, while having minimal impact in the under-committed case.

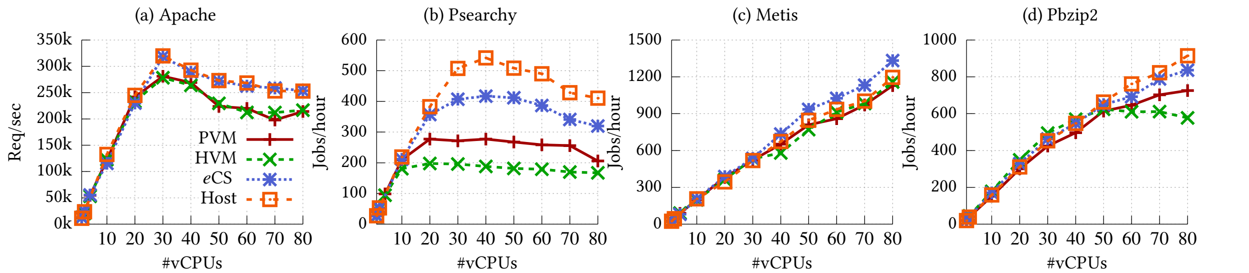


Figure 5: Performance of real-world workloads when running on the bare metal (Host), and inside a VM with three configurations: `PVM`, `HVM`, and with *e*CS annotations. In this scenario, only one VM is running. We use Host as the baseline for the comparison because we consider Host to have almost optimal performance.

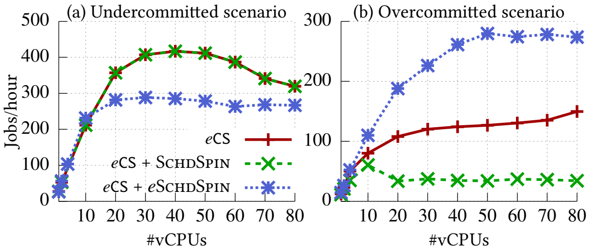


Figure 6: Impact of both `BWW` problem and *e*CS API (refer Hypersivor → VM in Table 1) on Psearchy in both under- and over-committed scenarios.

load that spends only around 5% of the time in the kernel space. Figure 3 (t:d) shows that the performance of *e*CS is similar to `PVM` and `HVM`, while decreasing the number of preemptions by 98.4–100% (refer (n:d)). We do not observe any performance gain in this scenario because 1) these preemptions may not be too critical to affect the application scalability, and 2) the overhead of our APIs, which do not provide any gains even after decreasing the preemptions. Similar to the other workloads, `LHP` dominates the preemption, followed by `RP`, `ICP`, and `RRP`.

In summary, our APIs not only reduce preemptions by 85–100%, but also improve the scalability of applications that use these synchronization primitives up to 2.5×, while no observable overhead on these applications. Moreover, we found that these preemptions occur for almost every type of primitives, specifically in the case of blocking synchronization primitives, read locks (Metis and Pbzip2), and interrupts (*e.g.*, TLB operations, packet processing etc.). In addition, most of the workloads still suffer from the `BWW` problem because of them being non-work conserving. We partially address this problem with the help of our *e*SchdSpin approach. One point to note is that we do not observe too many preemptions, as shown by prior works [42], because the current Linux kernel has dropped the FIFO-based Ticket `spinlock` and has replaced it with a highly optimized unfair queue-based lock [20] that mitigates the problem of `LHP` and `LWP`.

6.3 Performance in an Under-committed Case

We evaluate our *e*CS approach against `PVM` and `HVM` configurations in which a VM is running to show the impact of both APIs and *e*SchdSpin approach. We also include bare-metal configuration (Host) as a baseline (Figure 5). We observe that *e*CS addresses the `BWW` problem, and outperforms both `PVM` and `HVM` in the case of Apache (1.2× and 1.2×), Psearchy (1.6× and 1.9×), Metis (1.2× and 1.3×), and Psearchy (1.2× and 1.4×), while having almost similar latency for the Apache workload (Figure 4 (a)). Likewise, *e*CS performance is similar to that of bare-metal, except for the Psearchy workload.

For Apache, our APIs act as a back-off mechanism to improve its scalability, as the system is heavily contended. The throughput degrades after 30 cores because of the overhead of process scheduling, socket overhead, and inefficient kernel packet processing. Besides this, both Psearchy and Metis suffer from the `BWW` problem, which we improve with our *e*SchdSpin approach that results in better scalability as well as reduction in the idling of VMs. In particular, we decrease the idle time of Psearchy and Metis by 25% and 20%, respectively, by using our approach. One point to note is that blocking locks are based on the `TAS` lock, whose throughput severely degrades with increasing core count because of the increase cache-line contention, which we observe after 40 cores for Psearchy for all configurations. We also find that the Host is still 1.4× faster than *e*CS because *e*SchdSpin only partially mitigates the `BWW` problem, while introducing excessive cache-line contention, which we can circumvent with NUMA-aware locks [16]. For Pbzip2, we observe that *e*CS performs equivalent to the Host, while outperforming `PVM` and `HVM` after 60 cores, because Pbzip2 spends the least amount of time in the kernel space (5%), and starts to suffer from the `BWW` problem only after 60 cores, which our *e*SchdSpin easily tackles.

6.4 Addressing `BWW` Problem via *e*CS

We evaluate the impact of the `BWW` problem on Psearchy in both under- and over-committed scenarios. Figure 6 (a) shows that our scheduling-aware spinning approach (marked as *e*CS + SchdSpin) improves the throughput of Psearchy by 1.5× and 1.2× at 40 and 80 cores, respectively, in an under-committed scenario. SchdSpin approach allows a blocking waiter, both reader and writer, to actively spin for the lock if the number of tasks in the run queue is one, else the task schedules itself out. This approach is similar to the scheduling-aware parking/wake-up strategy [16], which we applied to the stock `mutex` and `rwsem`.

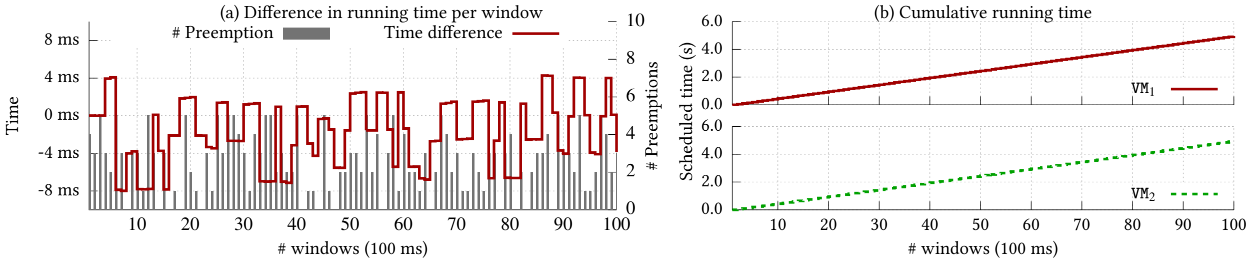


Figure 7: Fairness in *e*CS. Running time of a vCPU of two co-scheduled VMs (VM$_1$ and VM$_2$) with *e*CS annotations for a period of 10 seconds with 100 ms window granularity while executing a kernel intensive task (reading the contents of a file) that involves read side of rwsem. (a) shows the difference in running time of vCPU per window granularity as well as the number of preemptions occurring per window, while (b) illustrates the cumulative running time, and shows that the hypervisor maintains eventual fairness in the system, even if VM$_2$ is allowed extra schedules. Both VMs get 4.95 seconds to run.

As mentioned before, the reason for such an improvement is that the current design is not scheduling aware, as the waiter parks itself if it is unable to acquire the lock. With our approach, we try to mitigate this performance anomaly and allow the applications to scale further. Unfortunately, the scheduling-aware approach is inefficient in the case of the over-committed scenario, as shown in Figure 6 (b). The reason is that current waiters are guest OS agnostic, which leads to wasting CPU resources and resulting in more LHP and LWP problems, thereby degrading the scalability by almost 4.4× (marked *e*CS + SchdSpin in (b)) against a simple *e*CS configuration that still suffers from the BWW problem. We overcome this issue by using our `is_pcpu_overcommitted()` API that allows the SchdSpin approach to spin only when there is no active task on the pCPU's run queue; otherwise, the waiter is scheduled out when more than one task are in the run queue of the pCPU. By using our API (marked *e*CS + *e*SchdSpin), we outperform the baseline *e*CS approach by 1.8× and the *e*CS + SchdSpin approach by 8×.[3]

6.5 System Eventual Fairness

We now evaluate whether we are able to achieve eventual fairness while allowing *e*CS annotated VMs to obtain an extra schedule followed by local vCPU penalization. To evaluate the fairness, we run a simple micro-benchmark in two VMs (marked VM$_1$ and VM$_2$). VM$_1$ is a non-annotated VM, whereas VM$_2$ is an *e*CS annotated one. This micro-benchmark indefinitely reads the content of a file that stresses the read side of the rwsem and spends around 99% of the time in the kernel without scheduling out the task, thereby prohibiting the guest OS from doing any halt exits. Figure 7 (a) shows the time difference between two VM runtimes that we measure at every 100 ms window for each VM as well as the number of preemptions for VM$_2$ in that window. Figure 7 (b) shows the cumulative runtime of the VMs. We observe from Figure 7 (a) that even after allowing for extra schedules, the CFS scheduling policy balances out these extra schedules, which does not affect the runtime difference between VM$_1$ and VM$_2$. For example, at the end of one second window, marked 10, we observe that the number of extra schedules that the hypervisor granted VM$_2$ was 34 (34 milliseconds of extra time), but the runtime difference between VM$_1$ and VM$_2$ is 7.8 ms, which becomes -1.9 ms at the end of two seconds, while VM$_2$ received a total of 54 extra schedules (54 milliseconds). Hence, the extra schedule approach followed by our local vCPU penalization ensures that none of the tasks running on that particular physical CPU suffers from the fairness issue, also referred as eventual fairness. Moreover, Figure 7 (b) shows that both VMs get almost equivalent runtime in a lockstep fashion with both VMs getting almost 4.95 seconds at the end of 10 seconds.

7 Discussion

Our *e*CS approach addresses the problem of preemptions and BWW in both under- and over-committed scenarios by annotating all synchronization primitives and mechanisms in the kernel space. However, besides these primitives, kernel developers have to manually annotate a critical section if they want to avoid the preemptions while introducing their own primitives. One approach could be that the hypervisor can read the instruction pointer (IP) to figure out an *e*CS, but the guest OS must provide a guest OS symbol table to resolve the IP. In addition, the current design of *e*CS only targets the kernel space of a guest OS, and it is still agnostic of the user space critical sections such as pthread locks. Hence, we would like to extend our approach to the user space critical sections to further avoid the preemption problem, as we believe that *e*CS is a natural fit for multi-level scheduling. However, we need to communicate the scheduling hint down to the lowest layer effectively, which requires designing of the *e*CS composability extensions.

Our annotation approach does not open any security vulnerability because our approach is based on the paravirtualized VM, and it is similar to other approaches that share the information with the hypervisor [4, 19]. By using our virtualized scheduling-aware spinning ap-

[3] We have used *e*CS + *e*SchdSpin approach for our evaluation against PVM and HVM in §6.2 and §6.3.

proach (*e*SchdSpin), we partially mitigate the BWW problem. However, our Hypervisor → VM APIs expose scheduling information of the pCPU, but they only tell if a pCPU is overloaded or a vCPU is preempted. In addition, a VM cannot misuse this information as it will be later penalized by the hypervisor. There is also very slight possibility of priority inversion problem with our extra schedule approach. However, the window of that hypervisor-granted extra schedule is too small to incur priority inversion and performance, unlike co-scheduling approaches [41, 45] in which the scheduling window is in the order of several milliseconds.

8 Related work

The double scheduling phenomenon is a recurring problem in the domain of virtualization, which seriously impacts the performance of a VM. There have been comprehensive research efforts to mitigate this problem.

Synchronization primitives in VMs. Uhlig *et al.* [44] demonstrated the spinlock synchronization issue in a virtualized environment, which he addressed with synchronous hints to the hypervisor, and was later replaced by para-virtual hooks for the spinlock [8] for notifying the hypervisor to block the vCPU after it has exhausted its busy wait threshold. Meanwhile, other problems such as LWP [32], the BWW problem [5, 39], and RCU readers preemption problems were found. Gleaner [5] that addressed the BWW problem implemented a user space solution to handle tasks among a varying number of vCPUs, by manipulating tasks' processor affinity in the user space, which is difficult to maintain at runtime as it must accurately track each task launch and deletion. However, our *e*SchdSpin approach is user agnostic and mitigates the problem to certain extent for large core count.

Taebe *et al.* [42] addressed the LHP/LWP issue by exposing the time window from the hypervisor to the guest OS, which leverages this information that enables a waiter to either spin or join the waiting queue. However, their solution is not applicable to CFS [27] scheduler of Linux as it does not expose the scheduling window information. Their solution is orthogonal to our approach as we want the hypervisor to take a decision than the VM. Waiman Long [20] designed and implemented qspinlock that inherently overcomes the problem of LWP by exploiting the property of the TAS lock in the queue-based lock. It works by allowing the other waiters to steal the lock before joining the queue without disrupting the waiters' queue. However, qspinlock is still prone to LHP. Meanwhile, by annotating various locks as *e*CS, we confirm these problems, and further identify new sets of problems such as RP and ICP, and provide a simple solution to address the double scheduling phenomenon.

Partial handling of scheduling overhead in VMs. There have been several studies on virtualization overhead because of the software-hardware redirection [1, 39] and co-scheduling issues [17, 41, 45]. For example, VMware relies on relaxed co-scheduling [45] to mitigate double scheduling problem, in which vCPUs are scheduled in batches and the stragglers are synchronized within a predefined threshold. Besides this, other works have proposed balanced vCPU scheduling [41] or even IPI based demand scheduling [17]. However, these co-scheduling approaches suffer from CPU fragmentation. On the contrary, our approach neither introduces any CPU fragmentation nor it needs to synchronize the global scheduling information for all the vCPU of a VM because each vCPU is locally penalized by the hypervisor rather than synchronizing them among other vCPUs.

Song *et al.* [40] proposed the idea of dynamically adjusting vCPUs according to available CPU resources, while allowing guest OS to schedule its tasks. They used the approach of vCPU ballooning, which avoided the problem of double scheduling and was later extended by Cheng *et al.* [3] by designing a lightweight hotplug vCPU mechanism. Although their approach is effective in case of small VMs, it is complementary to our approach and may not scale effectively for large SMP VMs because of the overhead of migrating tasks from one vCPU to another as well as the frequent rescheduling of the targeted vCPUs. *e*CS, on the other hand, does not suffer from any explicit IPI and migration-specific tasks, as it only adds an overhead of a simple memory operations for a scheduling decision.

9 Conclusion

Double scheduling phenomenon is a well-known problem in the domain of virtualization that leads to several symptoms in the form of LHP, LWP, and BWW. We identify that it not only is limited to non-blocking locks, but also is applicable to blocking locks and reader side of locks. We present a single shot solution with our key insight: if a certain key component of a guest OS is allowed to proceed further, the guest OS will make forward progress. We identify these critical components as synchronization primitives and mechanism such as spinlocks, mutex, rwsem, RCU, and even interrupt context, which we call *enlightened critical sections* (*e*CS). We annotate *e*CS with our lightweight APIs that expose whether a VM is executing a critical section, which the hypervisor uses to provide an extra schedule at the scheduling boundary, thereby allowing the guest OS to progress forward. In addition, by leveraging the hypervisor scheduling context, a VM mitigates the effect of BWW problem with our simple virtualized spinning-aware spinning strategy. With *e*CS, we not only decrease the spurious preemptions by 85–100% but also improve the throughput of applications up to 1.6× and 2.5× in an under- and over-committed scenario, respectively.

References

[1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 2–13, New York, NY, USA, 2006. ACM.

[2] Amazon. Amazon Web Services, 2017. https://aws.amazon.com/.

[3] L. Cheng, J. Rao, and F. C. M. Lau. vScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, pages 2:1–2:14, London, UK, Apr. 2016. ACM.

[4] G. Costa. Steal time for KVM, 2011. https://lwn.net/Articles/449657/.

[5] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the Blocked-waiter Wakeup Problem for Virtualized Multicore Applications. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 73–84, Berkeley, CA, USA, 2014. USENIX Association.

[6] J. Fitzhardinge. Paravirtualized Spinlocks, 2008. http://lwn.net/Articles/289039/.

[7] T. A. S. Foundation. APACHE HTTP Server Project, 2017. https://httpd.apache.org/.

[8] T. Friebel. How to Deal with Lock-Holder Preemption. Technical report, Xen Summit, 2008.

[9] J. Gilchrist. Parallel BZIP2 (PBZIP2), Data Compression Software, 2017. http://compression.ca/pbzip2/.

[10] W. Glozer. wrk - a HTTP benchmarking tool, 2017. https://github.com/wg/wrk.

[11] Google. Compute Engine, 2017. https://aws.amazon.com/.

[12] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. Eli: Bare-metal performance for i/o virtualization. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS XVII, pages 411–422, London, UK, Mar. 2012. ACM.

[13] S. Kashyap. Vbench, 2015. https://github.com/sslab-gatech/vbench.

[14] S. Kashyap, C. Min, and T. Kim. Scalability In The Clouds! A Myth Or Reality? In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys)*, pages 5:1–5:7, New York, NY, USA, July 2015. ACM.

[15] S. Kashyap, C. Min, and T. Kim. Opportunistic Spinlocks: Achieving Virtual Machine Scalability in the Clouds. *SIGOPS Oper. Syst. Rev.*, 50(1):9–16, Mar. 2016.

[16] S. Kashyap, C. Min, and T. Kim. Scalable NUMA-aware Blocking Synchronization Primitives. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 603–615, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-38-6.

[17] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. Demand-based Coordinated Scheduling for SMP VMs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 369–380, New York, NY, USA, 2013. ACM.

[18] A. Kivity. sched: add notifier for process migration, 2009. https://lwn.net/Articles/356536/.

[19] W. Long. locking/qspinlock: Enhance pvqspinlock & introduce queued unfair lock, 2015. https://lwn.net/Articles/650776/.

[20] W. Long. qspinlock: a 4-byte queue spinlock with PV support, 2015. https://lkml.org/lkml/2015/4/24/631.

[21] Y. Mao, R. Morris, and F. M. Kaashoek. Optimizing MapReduce for Multicore Architectures. Technical report, MIT CSAIL, 2010.

[22] D. Matlack. Message Passing Workloads in KVM, 2015. http://www.linux-kvm.org/images/a/ac/02x03-Davit_Matalack-KVM_Message_passing_Performance.pdf.

[23] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991.

[24] Microsoft. SQL Server 2014, 2014. http://www.microsoft.com/en-us/server-cloud/products/sql-server/features.aspx.

[25] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 71–85, Denver, CO, June 2016. USENIX Association.

[26] I. Molnar. Linux rwsem, 2006. http://www.makelinux.net/ldd3/chp-5-sect-3.

[27] I. Molnar. [patch] Modular Scheduler Core and Completely Fair Scheduler [CFS], 2007. https://lwn.net/Articles/230501/.

[28] I. Molnar and D. Bueso. Generic Mutex Subsystem, 2016. https://www.kernel.org/doc/Documentation/locking/mutex-design.txt.

[29] I. Molnar and D. Bueso. Generic Mutex Subsystem, 2017. https://www.kernel.org/doc/Documentation/locking/mutex-design.txt.

[30] G. Naptov. KVM: Add asynchronous page fault for PV guest., 2009. https://lwn.net/Articles/359842/.

[31] O. Nesterov. Linux percpu-rwsem, 2012. http://lxr.free-electrons.com/source/include/linux/percpu-rwsem.h.

[32] J. Ouyang and J. R. Lange. Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 191–200, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1266-0.

[33] A. Prasad, K. Gopinath, and P. E. McKenney. The RCU-Reader Preemption Problem in VMs. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pages 265–270, Santa Clara, CA, July 2017. USENIX Association.

[34] M. Righini. Enabling Intel Virtualization Technology Features and Benefits, 2010.

[35] SAP. SAP HANA 2.0 SPS 02, 2017. http://hana.sap.com/abouthana.html.

[36] V. Seeker. Process Scheduling in Linux, 2013. https://www.prism-services.io/pdf/linux_scheduler_notes_final.pdf.

[37] A. Shi. [PATCH] rwsem: steal writing sem for better performance, 2013. https://lkml.org/lkml/2013/2/5/309.

[38] Q. Software. Demystifying CPU Ready (%RDY) as a Performance Metric, 2017. http://www.actualtechmedia.com/wp-content/uploads/2013/11/demystifying-cpu-ready.pdf.

[39] X. Song, H. Chen, and B. Zang. Characterizing the Performance and Scalability of Many-core Applications on Virtualized Platforms. Technical report, Fudan University, 2010.

[40] X. Song, J. Shi, H. Chen, and B. Zang. Schedule Processes, Not VCPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, pages 1:1–1:7, New York, NY, USA, 2013. ACM.

[41] O. Sukwong and H. S. Kim. Is Co-scheduling Too Expensive for SMP VMs? In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, pages 257–272, New York, NY, USA, Apr. 2011. ACM.

[42] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont. The Lock Holder and the Lock Waiter Pre-emption Problems: Nip Them in the Bud Using Informed Spinlocks (I-Spinlock). In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 286–297, New York, NY, USA, Apr. 2017. ACM.

[43] C.-C. Tu, M. Ferdman, C.-t. Lee, and T.-c. Chiueh. A Comprehensive Implementation and Evaluation of Direct Interrupt Delivery. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 1–15, New York, NY, USA, 2015. ACM.

[44] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[45] VMware. The CPU Scheduler in VMware ESX 4.1. Technical report, VMware, 2010.

[46] VMware. Best Practices for Oversubscription of CPU, Memory and Storage in vSphere Virtual Environments, 2017. https://communities.vmware.com/servlet/JiveServlet/previewBody/21181-102-1-28328/vsphere-oversubscription-best-practices%5b1%5d.pdf.

[47] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization

Yiming Zhang
NiceX Lab, NUDT

Jon Crowcroft
University of Cambridge

Dongsheng Li, Chengfei Zhang
NUDT

Huiba Li
Alibaba

Yaozheng Wang, Kai Yu
NUDT

Yongqiang Xiong
Microsoft

Guihai Chen
SJTU

Abstract

Unikernel specializes a minimalistic LibOS and a target application into a standalone single-purpose virtual machine (VM) running on a hypervisor, which is referred to as (virtual) *appliance*. Compared to traditional VMs, Unikernel appliances have smaller memory footprint and lower overhead while guaranteeing the same level of isolation. On the downside, Unikernel strips off the *process* abstraction from its monolithic appliance and thus sacrifices flexibility, efficiency, and applicability.

This paper examines whether there is a balance embracing the best of both Unikernel appliances (strong isolation) and processes (high flexibility/efficiency). We present KylinX, a dynamic library operating system for simplified and efficient cloud virtualization by providing the `pVM` (process-like VM) abstraction. A `pVM` takes the hypervisor as an OS and the Unikernel appliance as a process allowing both page-level and library-level dynamic mapping. At the page level, KylinX supports `pVM` fork plus a set of API for inter-`pVM` communication (IpC). At the library level, KylinX supports shared libraries to be linked to a Unikernel appliance at runtime. KylinX enforces mapping restrictions against potential threats. KylinX can fork a `pVM` in about 1.3 ms and link a library to a running `pVM` in a few ms, both comparable to process fork on Linux (about 1 ms). Latencies of KylinX IpCs are also comparable to that of UNIX IPCs.

1 Introduction

Commodity clouds (like EC2 [5]) provide a public platform where tenants rent virtual machines (VMs) to run their applications. These cloud-based VMs are usually dedicated to specific online applications such as big data analysis [24] and game servers [20], and are referred to as (virtual) appliances [56, 64]. The highly-specialized, single-purpose appliances need only a very small portion of traditional OS support to run their accommodated applications, while the current general-purpose OSs contain extensive libraries and features for multi-user, multi-application scenarios. The mismatch between the single-purpose usage of appliances and the general-purpose design of traditional OSs induces performance and security penalty, making appliance-based services cumbersome to deploy and schedule [62, 52], inefficient to run [56], and vulnerable to bugs of unnecessary libraries [27].

This problem has recently motivated the design of Unikernel [56], a library operating system (LibOS) architecture that is targeted for efficient and secure appliances in the clouds. Unikernel refactors a traditional OS into libraries, and seals the application binary and requisite libraries into a specialized appliance image which could run directly on a hypervisor such as Xen [30] and KVM [22]. Compared to traditional VMs, Unikernel appliances eliminate unused code, and achieve smaller memory footprint, shorter boot times and lower overhead while guaranteeing the same level of isolation [56]. The hypervisor's steady interface avoids hardware compatibility problems encountered by early LibOSs [39].

On the downside, Unikernel strips off the *process* abstraction from its statically-sealed monolithic appliances, and thus sacrifices flexibility, efficiency, and applicability. For example, Unikernel cannot support dynamic fork, a basis for commonly-used multi-process abstraction of conventional UNIX applications; and the compile-time determined immutability precludes runtime management such as online library update and address space randomization. This inability has largely reduced the applicability and performance of Unikernel.

In this paper, we examine whether there is a balance embracing the best of both Unikernel appliances (strong isolation) and processes (high flexibility/efficiency). We draw an analogy between appliances on a hypervisor and processes on a traditional OS and take one step forward from static Unikernels to present KylinX, a dynamic library operating system for simplified and efficient cloud virtualization by providing the `pVM` (process-like

VM) abstraction. We take the hypervisor as an OS and the appliance as a process allowing both page-level and library-level dynamic mapping for pVM.

At the page level, KylinX supports pVM fork plus a set of API for inter-pVM communication (IpC), which is compatible with conventional UNIX inter-process communication (IPC). The security of IpC is guaranteed by only allowing IpC between a *family* of mutually-trusted pVMs forked from the same root pVM.

At the library level, KylinX supports shared libraries to be dynamically linked to a Unikernel appliance, enabling pVMs to perform (i) online library update which replaces old libraries with new ones at runtime and (ii) recycling which reuses in-memory domains for fast booting. We analyze potential threats induced by dynamic mapping and enforce corresponding restrictions.

We have implemented a prototype of KylinX based on Xen [30] (a type-1 hypervisor) by modifying Mini-OS [14] (a Unikernel LibOS written in C) and Xen's toolstack. KylinX can fork a pVM in about 1.3 ms and link a library to a running pVM in a few ms, both comparable to process fork on Linux (about 1 ms). Latencies of KylinX IpCs are also comparable to that of UNIX IPCs. Evaluation on real-world applications (including a Redis server [13] and a web server [11]) shows that KylinX achieves higher applicability and performance than static Unikernels while retaining the isolation guarantees.

The rest of this paper is organized as follows. Section 2 introduces the background and design options. Section 3 presents the design of dynamically-customized KylinX LibOS with security restrictions. Section 4 reports the evaluation results of the KylinX prototype implementation. Section 5 introduces related work. And Section 6 concludes the paper and discusses future work.

2 Preliminaries

2.1 VMs, Containers & Picoprocesses

There are several conventional models in the literature of virtualization and isolation: processes, Jails, and VMs.

- OS processes. The process model is targeted for a conventional (partially-trusted) OS environment, and provides rich ABI (application binary interface) and interactivity that make it not suitable for truly adversarial tenants.
- FreeBSD Jails [47]. The jail model provides a lightweight mechanism to separate applications and their associated policies. It runs a process on a conventional OS, but restricts several of the syscall interfaces to reduce vulnerability.
- VMs. The VM model builds an isolation boundary matching hardware. It provides legacy compatibility for guests to run a complete OS, but it is costly due to duplicated and vestigial OS components.

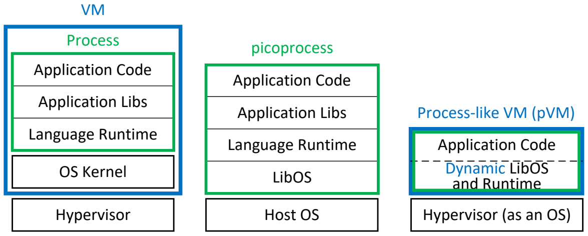


Figure 1: Alternative virtualization architectures.

VMs (Fig. 1(*left*)) have been widely used in multi-tenant clouds since it guarantees strong (type-1-hypervisor) isolation [55]. However, the current virtualization architecture of VMs is heavy with layers of hypervisor, VM, OS kernel, process, language runtime (such as glibc [16] and JVM [21]), libraries, and application, which are complex and could no longer satisfy the efficiency requirements of commercial clouds.

Containers (like LXC [9] and Docker [15]) leverage kernel features to package and isolate processes. They are recently in great demand [25, 7, 6] because they are lightweight compared to VMs. However, containers offer weaker isolation than VMs, and thus they often run in VMs to achieve proper security guarantees [58].

Picoprocesses [38] (Fig. 1 (*center*)) could be viewed as containers with stronger isolation but lighter-weight host obligations. They use a small interface between the host OSs and the guests to implement a LibOS realizing the host ABI and map high-level guest API onto the small interface. Picoprocesses are particularly suitable for client software delivery because client software needs to run on various host hardware and OS combinations [38]. They could also run on top of hypervisors [62, 32].

Recent studies [67, 32, 54] on picoprocesses relax the original static isolation model by allowing dynamics. For example, Graphene [67] supports picoprocess fork and multi-picoprocess API, and Bascule [32] allows OS-independent extensions to be attached to a picoprocess at runtime. Although these relaxations dilute the strict isolation model, they effectively extend the applicability of picoprocesses to a much broader range of applications.

2.2 Unikernel Appliances

Process-based virtualization and isolation techniques face challenges from the broad kernel syscall API that is used to interact with the host OS for, e.g., process/thread management, IPC, networking, etc. The number of Linux syscalls has reached almost 400 [3] and is continuously increasing, and the syscall API is much more difficult to secure than the ABI of VMs (which could leverage hardware memory isolation and CPU rings) [58].

Recently, researchers propose to reduce VMs, instead of augmenting processes, to achieve secure and efficient cloud virtualization [56, 19, 49]. Unikernel [56] is focused on single-application VM appliances [26] and adapts the Exokernel [39] style LibOS to VM guests to enjoy performance benefits while preserving the strong isolation guarantees of a type-1 hypervisor. It breaks the traditional general-purpose virtualization architecture (Fig. 1 (*left*)) and implements the OS features (e.g., device drivers and networking) as libraries. Compared to other hypervisor-based reduced VMs (like Tiny Core Linux [19] and OS^v [49]), Unikernel seals only the application and its requisite libraries into the image.

Since the hypervisor already provides a number of management features (such as isolation and scheduling) of traditional OSs, Unikernel adopts the *minimalism* philosophy [36], which minimizes the VMs by not only removing unnecessary libraries but also stripping off the duplicated management features from its LibOS. For example, Mirage [57] follows the multikernel model [31] and leverages the hypervisor for multicore scheduling, so that the single-threaded runtime could have fast sequential performance; MiniOS [14] relies on the hypervisor (instead of an in-LibOS linker) to load/link the appliance at boot time; and LightVM [58] achieves fast VM boot by redesigning Xen's control plane.

2.3 Motivation & Design Choices

Unikernel appliances and conventional UNIX processes both abstract the unit of isolation, privileges, and execution states, and provide management functionalities such as memory mapping, execution cooperation, and scheduling. To achieve low memory footprint and small trusted computing base (TCB), Unikernel strips off the process abstraction from its monolithic appliance and links a minimalistic LibOS against its target application, demonstrating the benefit of relying on the hypervisor to eliminate duplicated features. But on the downside, the lack of processes and compile-time determined monolithicity largely reduce Unikernel's flexibility, efficiency, and applicability.

As shown in Fig. 1 (*right*), KylinX provides the `pVM` abstraction by explicitly taking the hypervisor as an OS and the Unikernel appliance as a process. KylinX slightly relaxes Unikernel's compile-time monolithicity requirement to allow both page-level and library-level dynamic mapping, so that `pVMs` could embrace the best of both Unikernel appliances and UNIX processes. As shown in Table 1, KylinX could be viewed as an extension (providing the `pVM` abstraction) to Unikernel, similar to the extention of Graphene [67] (providing conventional multi-process compatibility) and Bascule [32] (providing runtime extensibility) to picoprocess.

	Static	**Dynamic**
Picoprocess	Embassies [43], Xax [38], etc.	Graphene [67], Bascule [32], etc
Unikernel	Mirage [57], MiniOS [14], etc.	**KylinX**

Table 1: Inspired by dynamic picoprocesses, KylinX explores new design space and extends the applicability of Unikernel.

We implement KylinX's dynamic mapping extension in the hypervisor instead of the guest LibOS for the following reasons. First, an extension outside the guest LibOS allows the hypervisor to enforce mapping restrictions (§3.2.3 and §3.3.4) and thus improves security. Second, the hypervisor is more flexible to realize dynamic management for, e.g., restoring live states during `pVM`'s online library update (§3.3.2). And third, it is natural for KylinX to follow Unikernel's minimalism philosophy (§2.2) of leveraging the hypervisor to eliminate duplicated guest LibOS features.

Backward compatibility is another tradeoff. The original Mirage Unikernel [56] takes an extreme position where existing applications and libraries have to be completely rewritten in OCaml [10] for type safety, which requires a great deal of engineering effort and may introduce new vulnerabilities and bugs. In contrast, KylinX aims to support source code (mainly `C`) compatibility, so that a large variety of legacy applications could run on KylinX with minimum effort for adaptation.

Threat model. KylinX assumes a traditional threat model [56, 49], the same context as Unikernel [56] where VMs/`pVMs` run on the hypervisor and are expected to provide network-facing services in a public multi-tenant cloud. We assume the adversary can run untrusted code in the VMs/`pVMs`, and applications running in the VMs/`pVMs` are under potential threats both from other tenants in the same cloud and from malicious hosts connected via Internet. KylinX treats both the hypervisor (with its toolstacks) and the control domain (dom0) as part of the TCB, and leverages the hypervisor for isolation against attacks from other tenants. The use of secure protocols like SSL and SSH helps KylinX `pVMs` trust external entities.

Recent advance in hardware like Intel Software Guard eXtensions (SGX) [12] demonstrates the feasibility of shielded execution in *enclaves* to protect VMs/`pVMs` from the privileged hypervisor and dom0 [33, 28, 45], which will be studied in our future work. We also assume hardware devices are not compromised, although in rare cases hardware threats have been identified [34].

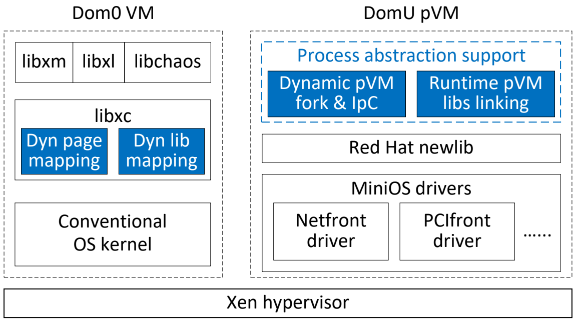


Figure 2: KylinX components. Blue parts are newly designed. DomU pVM is essentially a Unikernel appliance.

3 KylinX Design

3.1 Overview

KylinX extends Unikernel to realize desirable features that are previously applicable only to processes. Instead of designing a new LibOS from scratch, we base KylinX on MiniOS [27], a C-style Unikernel LibOS for user VM domains (domU) running on the Xen hypervisor. MiniOS uses its front-end drivers to access hardware, which connect to the corresponding back-end drivers in the privileged dom0 or a dedicated driver domain. MiniOS has a single address space without kernel and user space separation, as well as a simple scheduler without preemption. MiniOS is tiny but fits the bill allowing a neat and efficient LibOS design on Xen. For example, Erlang on Xen [1], LuaJIT [2], ClickOS [59] and LightVM [58] leverage MiniOS to provide Erlang, Lua, Click and fast boot environments, respectively.

As shown in Fig. 2, the MiniOS-based KylinX design consists of (i) the (restricted) dynamic page/library mapping extensions of Xen's toolstack in Dom0, and (ii) the process abstraction support (including dynamic pVM fork/IpC and runtime pVM library linking) in DomU.

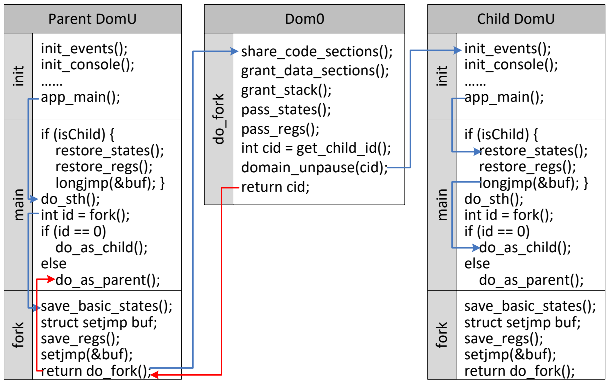
3.2 Dynamic Page Mapping

KylinX supports process-style appliance fork and communication by leveraging Xen's shared memory and grant tables to perform cross-domain page mapping.

3.2.1 pVM Fork

The fork API is the basis for realizing traditional multiprocess abstractions for pVMs. KylinX treats each user domain (pVM) as a process, and when the application invokes fork() a new pVM will be generated.

We leverage the memory sharing mechanism of Xen to implement the fork operation, which creates a *child* pVM by (i) duplicating the xc_dom_image structure and (ii) invoking Xen's unpause() API to fork the calling parent pVM and return its domain ID to the parent. As shown in Fig. 3, when fork() is invoked in the parent pVM, we use inline assemblies to get the current states of CPU registers and pass them to the child. The control domain (dom0) is responsible for forking and starting the child pVM. We modify *libxc* to keep the xc_dom_image structure in memory when the parent pVM was created, so that when fork() is invoked the structure could be directly mapped to the virtual address space of the child pVM and then the parent could share sections with the child using grant tables. Writable data is shared in a copy-on-write (CoW) manner.



```
init_events();
init_console();
......
app_main();

if (isChild) {
   restore_states();
   restore_regs();
   longjmp(&buf); }
do_sth();
int id = fork();
if (id == 0)
   do_as_child();
else
   do_as_parent();

save_basic_states();
struct setjmp buf;
save_regs();
setjmp(&buf);
return do_fork();
```

```
share_code_sections();
grant_data_sections();
grant_stack();
pass_states();
pass_regs();
int cid = get_child_id();
domain_unpause(cid);
return cid;
```

```
init_events();
init_console();
......
app_main();

if (isChild) {
   restore_states();
   restore_regs();
   longjmp(&buf); }
do_sth();
int id = fork();
if (id == 0)
   do_as_child();
else
   do_as_parent();

save_basic_states();
struct setjmp buf;
save_regs();
setjmp(&buf);
return do_fork();
```

Figure 3: pVM fork. After *fork* is invoked, KylinX creates a child pVM by sharing the parent (caller) pVM's pages.

After the child pVM is started via unpause(), it (i) accepts the shared pages from its parent, (ii) restores the CPU registers and jumps to the next instruction after fork, and (iii) begins to run as a child. After fork() is completed, KylinX asynchronously initializes an event channel and shares dedicated pages between the parent and child pVMs to enable their IpC, as introduced in the next subsection.

3.2.2 Inter-pVM Communication (IpC)

KylinX provides a multi-process (multi-pVM) application with the view that all of its processes (pVMs) are collaboratively running on the OS (hypervisor). Currently KylinX follows the strict isolation model [67] where only mutually-trusted pVMs can communicate with each other, which will be discussed in more details in §3.2.3.

The two communicating pVMs use an event channel and shared pages to realize inter-pVM communication. If two mutually-trusted pVMs have not yet initialized an event channel when they communicate for the first time because they have no parent-child relationship via fork() (§3.2.1), then KylinX will (i) verify their mutual trustworthiness (§3.2.3), (ii) initialize an event channel, and (iii) share dedicated pages between them.

Type	API	Description
Pipe	pipe	Create a pipe and return the fds.
	write	Write *value* to a pipe.
	read	Read *value* from a pipe.
Signal	kill	Send signal to a domain.
	exit	Child sends SIGCHLD to parent.
	wait	Parent waits for child's signal.
Message Queue	ftok	Return the key for a given *path*.
	msgget	Create a message queue for *key*.
	msgsnd	Write *msg* to message queue.
	msgrcv	Read *msg* from message queue.
Shared Memory	shmget	Create & share a memory region.
	shmat	Attach shared memory (of *shmid*).
	shmdt	Detach shared memory.

Table 2: Inter-pVM communication API.

The event channel is used to notify events, and the shared pages are used to realize the communication. KylinX has already realized the following four types of inter-pVM communication APIs (listed in Table. 2).

(1) pipe(fd) creates a pipe and returns two file descriptors (fd[0] and fd[1]), one for write and the other for read.

(2) kill(domid, SIG) sends a signal (SIG) to another pVM (domid) by writing SIG to the shared page and notifying the target pVM (domid) to read the signal from that page; exit and wait are implemented using kill.

(3) ftok(path, projid) translates the path and projid to an IpC key, which will be used by msgget(key, msgflg) to create a message queue with the flag (msgflg) and return the queue ID (msgid); msgsend(msgid, msg, len) and msgrcv(msgid, msg, len) write/read the queue (msgid) to/from the msgbuf structure (msg) with length len.

(4) shmget(key, size, shmflg) creates and shares a memory region with the key (key), memory size (size) and flag (shmflg), and returns the shared memory region ID (shmid), which could be attached and detached by shmat(shmid, shmaddr, shmflg) and shmdt(shmaddr).

3.2.3 Dynamic Page Mapping Restrictions

When performing dynamic pVM fork, the parent pVM shares its pages with an empty child pVM, the procedure of which introduces no new threats.

When performing IpC, KylinX guarantees the security by the abstraction of a *family* of mutually-trusted pVMs, which are forked from the same root pVM. For example, if a pVM *A* forks a pVM *B*, which further forks another pVM *C*, then the three pVMs *A*, *B*, and *C* belong to the same family. For simplicity, currently KylinX follows the all-all-nothing isolation model: only the pVMs belonging to the same family are considered to be trusted and are allowed to communicate with each other. KylinX rejects communication requests between untrusted pVMs.

3.3 Dynamic Library Mapping

3.3.1 pVM Library Linking

Inherited from MiniOS, KylinX has a single flat virtual memory address space where application binary and libraries, system libraries (for bootstrap, memory allocation, etc.), and data structures co-locate to run. KylinX adds a *dynamic segment* into the original memory layout of MiniOS, so as to accommodate dynamic libraries after they are loaded.

As depicted in Fig. 2, we implement the dynamic library mapping mechanism in the Xen control library (*libxc*), which is used by the upper-layer toolstacks such as *xm/xl/chaos*. A pVM is actually a para-virtualized domU, which (i) creates a domain, (ii) parses the kernel image file, (iii) initializes the boot memory, (iv) builds the image in the memory, and (v) boots up the image for domU. In the above 4th step, we add a function (xc_dom_map_dyn()) to map the shared libraries into the *dynamic segment*, by extending the static linking procedure of *libxc* as follows.

- First, KylinX reads the addresses, offsets, file sizes and memory sizes of the shared libraries from the program header table of the appliance image.
- Second, it verifies whether the restrictions (§3.3.4) are satisfied. If not, the procedure terminates.
- Third, for each dynamic library, KylinX retrieves the information of its dynamic sections including the dynamic string table, symbol table, etc.
- Fourth, KylinX maps all the requisite libraries throughout the dependency tree into the dynamic segment of the pVM, which will *lazily* relocate an unresolved symbol to the proper virtual address when it is actually accessed.
- Finally, it jumps to the pVM's entry point.

KylinX will not load/link the shared libraries until they are actually used, which is similar to lazy binding [17] for conventional processes. Therefore, the boot times of KylinX pVMs are lower than that of previous Unikernel VMs. Further, compared to previous Unikernels which support only static libraries, another advantage of KylinX using shared libraries is that it effectively reduces the memory footprint in high-density deployment (e..g., 8K VMs per machine in LightVM [58] and 80K containers per machine in Flurries [71]), which is the single biggest factor [58] limiting both scalability and performance.

Next, we will discuss two simple applications of dynamic library mapping of KylinX pVMs.

3.3.2 Online pVM Library Update

It is important to keep the system/application libraries up to date to fix bugs and vulnerabilities. Static Unikernel [56] has to recompile and reboot the entire appliance image to apply updates for each of its libraries, which may result in significant deployment burdens when the appliance has many third-party libraries.

Online library update is more attractive than rolling reboots mainly in *keeping connections* to the clients. First, when the server has many long-lived connections, rebooting will result in high reconnection overhead. Second, it is uncertain whether a third-party client will re-establish the connections or not, which imposes complicated design logic for reconnection after rebooting. Third, frequent rebooting and reconnection may severely degrade the performance of critical applications such as high-frequency trading.

Dynamic mapping makes it possible for KylinX to realize online library update. However, libraries may have their own states for, e.g., compression or cryptography, therefore simply replacing stateless functions cannot satisfy KylinX's requirement.

Like most library update mechanisms (including DYMOS [51], Ksplice [29], Ginseng [61], PoLUS [37], Katana [63], Kitsune [41], etc), KylinX requests the new and old libraries to be binary-compatible: it is allowed to add new functions and variables to the library, but it is not allowed to change the interface of functions, remove functions/variables, or change fields of structures. For library states, we expect all the states are stored as variables (or dynamically-allocated structures) that would be saved and restored during update.

KylinX provides the `update(domid, new_lib, old_lib)` API to dynamically replace `old_lib` with `new_lib` for a domU `pVM` (ID = `domid`), with necessary update of library states. We also provide an *update* command "`update domid, new_lib, old_lib`" for parsing parameters and calling the `update()` API.

The difficulty of dynamic `pVM` update lies in manipulating symbol tables in a sealed VM appliance. We leverage dom0 to address this problem. When the `update` API is called, dom0 will (i) map the new library into dom0's virtual address space; (ii) share the loaded library with domU; (iii) verify whether the old library is quiescent by asking domU to check the call stack of each kernel thread of domU; (iv) wait until the old library is not in use and pause the execution; (v) modify the entries of affected symbols to the proper addresses; and finally (vi) release the old library. In the above 5th step, there are two kinds of symbols (*functions* and *variables*) which will be resolved as discussed below.

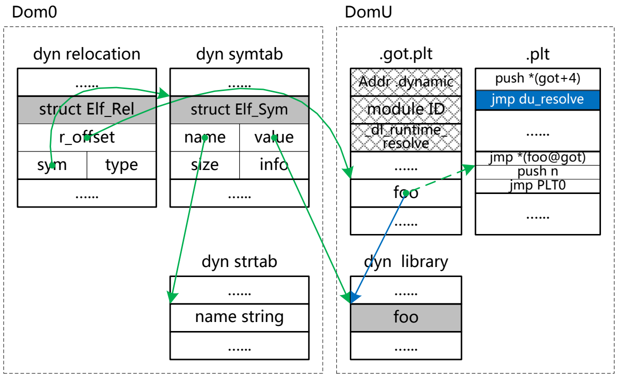


Figure 4: KylinX dynamic symbol resolution for functions. The green lines represent pointers for normal lazy binding of processes. The blue line represents the result of KylinX's resolution, pointing to the real function instead of the .plt table entry (dashed green line).

Functions. The dynamic resolution procedure for functions is illustrated in Fig. 4. We keep the relocation table, symbol table and string table in dom0 as they are not in the loadable segments. We load the global offset table of functions (.got.plt) and the procedure linkage table (.plt) in dom0 and share them with domU. In order to resolve symbols across different domains, we modify the 2nd line of assembly in the 1st entry of the .plt table (as shown in the blue region in Fig. 4) to point to KylinX's symbol resolve function (`du_resolve`). After the new library (`new_lib`) is loaded, the entry of each function of `old_lib` in the .got.plt table (e.g., `foo` in Fig. 4) is modified to point to the corresponding entry in the .plt table, i.e., the 2nd assembly (`push n`) shown by the dashed green line in Fig. 4. When a function (`foo`) of the library is called for the first time after `new_lib` is loaded, `du_resolve` will be called with two parameters (`n` and `*(got+4)`), where `n` is the offset of the symbol (`foo`) in the .got.plt table, and `*(got+4)` is the ID of the current module. `du_resolve` then asks dom0 to call its counterpart `d0_resolve`, which finds `foo` in `new_lib` and updates the corresponding entry (located by `n`) in the .got.plt table of the current module (ID = `module_ID`) to the proper address of `foo` (the blue line in Fig. 4).

Variables. Dynamic resolution for variables is slightly complex. Currently we simply assume that `new_lib` expects all its variables to be set to their live states in `old_lib` instead of their initial values. Without this restriction, the compiler will need extensions to allow developers to specify their intention for each variable.

(1) Global variables. If a global variable (g) of the library is accessed in the main program, then g is stored in the data segment (.bss) of the program and there is

an entry in the global offset table (.got) of the library pointing to g, so after new_lib is loaded KylinX will resolve g's entry in the .got table of new_lib to the proper address of g. Otherwise, g is stored in the data segment of the library and so KylinX is responsible for copying the global variable g from old_lib to new_lib.

(2) Static variables. Since static variables are stored in the data segment of the library and cannot be accessed from outside, after new_lib is loaded KylinX will simply copy them one by one from old_lib to new_lib.

(3) Pointers. If a library pointer (p) points to a dynamically-allocated structure, then KylinX preserves the structure and set p in new_lib to it. If p points to a global variable stored in the data segment of the program, then p will be copied from old_lib to new_lib. If p points to a static variable (or a global variable stored in the library), then p will point to the new address.

3.3.3 pVM Recycling

The standard boot (§3.3.1) of KylinX pVMs and Unikernel VMs [58] is relatively slow. As evaluated in §4.1, it takes 100+ ms to boot up a pVM or a Unikernel VM, most time of which is spent in creating the empty domain. Therefore, we design a pVM recycling mechanism for KylinX pVMs which leverages dynamic library mapping to bypass domain creation.

The basic idea of recycling is to reuse an in-memory empty domain to dynamically map the application (as a shared library) to that domain. Specifically, an empty recyclable domain is checkpointed and waits for running an application before calling the app_entry function of a *placeholder* dynamic library. The application is compiled into a shared library instead of a bootable image, using app_entry as its entry. To accelerate the booting of a pVM for the application, KylinX restores the checkpointed domain, and links the application library by replacing the placeholder library following the online update procedure (§3.3.2).

3.3.4 Dynamic Library Mapping Restrictions

KylinX should isolate any new vulnerabilities compared to the statically and monolithically sealed Unikernel when performing dynamic library mapping. The main threat is that the adversary may load a malicious library into the pVM's address space, replace a library with a compromised one that has the same name and symbols, or modify the entries in the symbol table of a shared library to the fake symbols/functions.

To address these threats, KylinX enforces restrictions on the identities of libraries as well as the loaders of the libraries. KylinX supports developers to specify the restrictions on the signature, version, and loader of the dynamic library, which are stored in the header of the pVM image and will be verified before linking a library.

Signature and version. The library developer first generates the library's SHA1 digest that will be encrypted by RSA (Rivest-Shamir-Adleman). The result is saved in a *signature* section of the dynamic library. If the appliance requires signature verification of the library, the signature section will be read and verified by KylinX using the public key. Version restrictions are requested and verified similarly.

Loader. The developer may request different levels of restrictions on the loader of the libraries: (i) only allowing the pVM itself to be the loader; (ii) also allowing other pVMs of the same application; or (iii) even allowing pVMs of other applications. With the first two restrictions a malicious library in one compromised application would not affect others. Another case for loader check is to load the application binary as a library and link it against a pVM for fast recycling (§3.3.3), where KylinX restricts the loader to be an empty pVM.

With these restrictions, KylinX introduces no new threats compared to the statically-sealed Unikernel. For example, runtime library update (§3.3.2) of a pVM with restrictions on the signature (to be the trusted developer), version (to be the specific version number), and loader (to be the pVM itself) will have the same level of security guarantees as recompiling and rebooting.

4 Evaluation

We have implemented a prototype of KylinX on top of Ubuntu 16.04 and Xen. Following the default settings of MiniOS [14], we respectively use RedHat Newlib and lwIP as the libc/libm libraries and TCP/IP stack. Our testbed has two machines each of which has an Intel 6-core Xeon E5-2640 CPU, 128 GB RAM, and one 1GbE NIC.

We have ported a few applications to KylinX, among which we will use a multi-process Redis server [13] as well as a multi-thread web server [11] to evaluate the application performance of KylinX in §4.6. Due to the limitation of MiniOS and RedHat Newlib, currently two kinds of adaptations are necessary for porting applications to KylinX. First, KylinX can support only select but not the more efficient epoll. Second, inter-process communications (IPC) are limited to the API listed in Table 2.

4.1 Standard Boot

We evaluate the time of the standard boot procedure (§3.3.1) of KylinX pVMs, and compare it with that of MiniOS VMs and Docker containers, all running a Redis

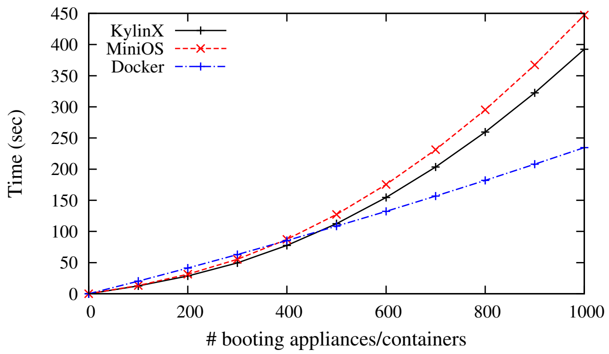


Figure 5: Total time of standard booting (reduced Redis).

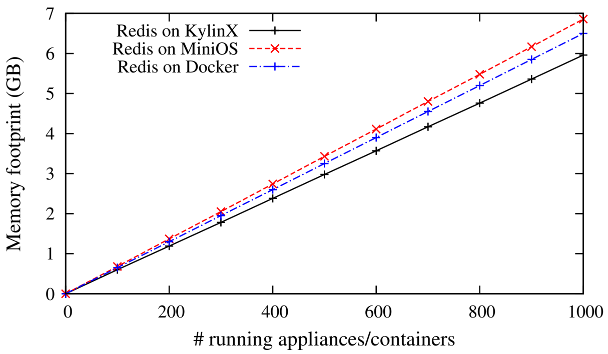


Figure 6: Memory usage (reduced Redis).

server. Redis is an in-memory key-value store that supports fast key-value storage/queries. Each key-value pair consists of a fixed-length key and a variable-length value. It uses a single-threaded process to serve user requests, and realizes (periodic) serialization by forking a new backup process.

We disable XenStore logging to eliminate the interference of periodic log file flushes. The C library (*libc*) of RedHat Newlib is static for use in embedded systems and difficult to be converted into a shared library. For simplicity, we compile *libc* into a static library and *libm* (the math library of Newlib) into a shared library that will be linked to the KylinX `pVM` at runtime. Since MiniOS cannot support `fork`, we (temporarily) remove the corresponding code in this experiment.

It takes about 124 ms to boot up a single KylinX `pVM` which could be roughly divided into two stages, namely, creating the domain/image in memory (steps 1 $\sim$ 4 in §3.3.1), and booting the image (step 5). Dynamic mapping is performed in the first stage. Most of the time (about 121 ms) is spent in the first stage, which invokes hypercalls to interact with the hypervisor. The second stage takes about 3 ms to start the `pVM`. In contrast, MiniOS takes about 133 ms to boot up a VM, and Docker takes about 210 ms to start a container. KylinX takes less time than MiniOS mainly because its shared libraries are not read/linked during the booting.

We then evaluate the total times of sequentially booting up a large number (up to 1K) of `pVMs` on one machine. We also evaluate the total boot times of MiniOS VMs and Docker containers for comparison.

The result is depicted in Fig. 5. First, KylinX is slightly faster than MiniOS owing to its lazy loading/linking. Second, the boot times of both MiniOS and KylinX increase superlinearly as the number of VMs/`pVMs` increases while the boot time of Docker containers increases only linearly, mainly because XenStore is highly inefficient when serving a large number of VMs/`pVMs` [58].

4.2 Fork & Recycling

Compared to containers, KylinX's standard booting cannot scale well for a large number of `pVMs` due to the inefficient XenStore. Most recently, LightVM [58] completely redesigns Xen's control plane by implementing chaos/libchaos, `noxs` (no XenStore), and split toolstack, together with a number of other optimizations, so as to achieve ms-level booting times for a large number of VMs. We adopt LightVM's `noxs` for eliminating XenStore's affect and test the `pVM` fork mechanism running *unmodified* Redis emulating conventional process fork. LightVM's `noxs` enables the boot times of KylinX `pVMs` to increase linearly even for a large number of `pVMs`. The fork of a single `pVM` takes about 1.3 ms (not shown here due to lack of space), several times faster than LightVM's original boot procedure (about 4.5 ms). KylinX `pVM` fork is slightly slower than a process fork (about 1 ms) on Ubuntu, because several operations including page sharing and parameter passing are time-consuming. Note that the initialization for the event channel and shared pages of parent/child `pVMs` is asynchronously performed and thus does not count for the latency of fork.

4.3 Memory Footprint

We measure the memory footprint of KylinX, MiniOS and Docker (Running Redis) for different numbers of `pVMs`/VMs/containers on one machine. The result (depicted in Fig. 6) proves that KylinX `pVMs` have smaller memory footprint compared to statically-sealed MiniOS and Docker containers. This is because KylinX allows the libraries (except *libc*) to be shared by all appliances of the same application (§3.3), and thus the shared libraries need to be loaded at most once. The memory footprint advantage facilitates ballooning [42] which could be used to dynamically share physical memory between VM appliances, and enables KylinX to achieve comparable memory efficiency with page-level deduplication [40] while introducing much less complexity.

	pipe	msg_que	kill	exit/wait	sh_m
KylinX[1]	55	43	41	43	39
KylinX[2]	240	256	236	247	232
Ubuntu	54	97	68	95	53

Table 3: IpC vs. IPC in latency (μs). KylinX[1]: a pair of lineal pVMs which already have an event channel and shared pages. KylinX[2]: a pair of non-lineal pVMs.

4.4 Inter-pVM Communication

We evaluate the performance of inter-pVM communication (IpC) by forking a parent pVM and measuring the parent/child communication latencies. We refer to a pair of parent/child pVMs as *lineal* pVMs. As introduced in §3.2.1, two lineal pVMs already have an event channel and shared pages and thus they could communicate with each other directly. In contrast, non-lineal pVM pairs have to initialize the event channel and shared pages before their first communication.

The result is listed in Table 3, and we compare it with that of the corresponding IPCs on Ubuntu. KylinX IpC latencies between two lineal pVMs are comparable to the corresponding IPC latencies on Ubuntu, owing to the high-performance event channel and shared memory mechanism of Xen. Note that the latency of pipe includes not only creating a pipe but also writing and reading a value through the pipe. The first-time communication latencies between non-lineal pVMs are several times higher due to the initialization cost.

4.5 Runtime Library Update

We evaluate runtime library update of KylinX by dynamically replacing the default *libm* (of RedHat Newlib 1.16) with a newer version (of RedHat Newlib 1.18). *libm* is a math library used by MiniOS/KylinX and contains a collection of 110 basic math functions.

To test KylinX's update procedure for global variables, we also add 111 pseudo global variables as well as one read_global function (reading out all the global variables) to both the old and the new *libm* libraries. The main function first sets the global variables to random values and then periodically verifies these variables by calling the read_global function.

Consequently, there are totally 111 functions as well as 111 variables that need to be updated in our test. The update procedure could be roughly divided into 4 stages and we measure the time of each stage's execution.

First, KylinX loads new_lib into the memory of dom0 and shares it with domU. Second, KylinX modifies the relevant entries of the functions in the .got.plt table to point to the corresponding entries in the .plt table. Third, KylinX calls du_resolve for each of the functions which asks dom0 to resolve the given function and returns its address in new_lib, and then updates the corresponding entries to the returned addresses. Finally, KylinX resolves the corresponding entries of the global variables in the .got table of new_lib to the proper addresses. We modify the third stage in our evaluation to update all the 111 functions in *libm* at once, instead of lazily linking a function when it is actually being called (§3.3.2), so as to present an overview of the entire runtime update cost of *libm*.

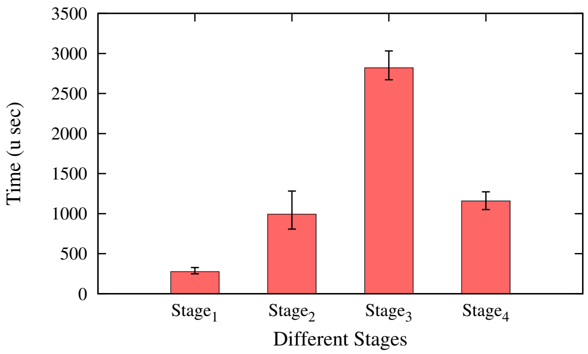


Figure 7: Runtime library update.

The result is depicted in Fig. 7, where the total overhead for updating all the functions and variables is about 5 milliseconds. The overhead of the third stage (resolving functions) is higher than others including the fourth stage (resolving variables), which is caused by several time-consuming operations in the third stage including resolving symbols, cross-domain invoking d0_resolve, returning real function addresses and updating corresponding entries.

4.6 Applications

Besides the process-like flexibility and efficiency of pVM scheduling and management, KylinX also provides high performance for its accommodated applications comparable to that of their counterparts on Ubuntu, as evaluated in this subsection.

4.6.1 Redis Server Application

We evaluate the performance of Redis server in a KylinX pVM, and compare it with that in MiniOS/Ubuntu. Again, since MiniOS cannot support fork(), we temporarily remove the code for serialization. The Redis server uses select instead of epoll to realize asynchronous I/O, because epoll is not yet supported by the lwIP stack [4] used by MiniOS and KylinX.

We use the Redis benchmark [13] to evaluate the performance, which uses a configurable number of busy loops asynchronously writing KVs. We run different

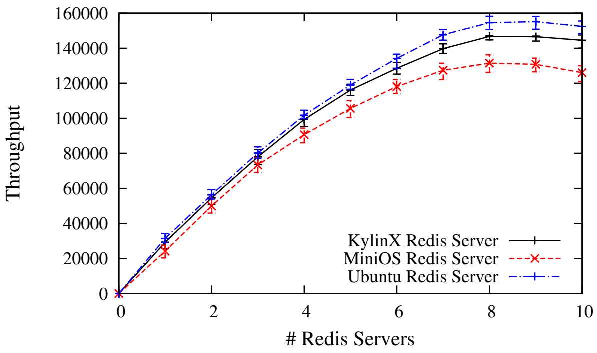


Figure 8: Redis server application.

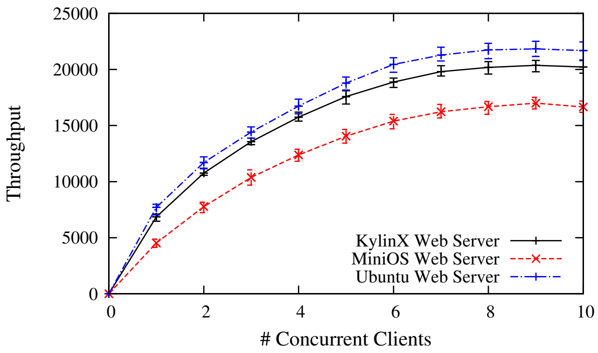


Figure 9: Web server application.

numbers of pVMs/VMs/processes (each for 1 server) servicing write requests from clients. We measure the write throughput as a function of the number of servers (Fig. 8). The three kinds of Redis servers have similar write throughput (due to the limitation of `select`), increasing almost linearly with the numbers of concurrent servers (scaling being linear up to 8 instances before the lwIP stack becomes the bottleneck).

4.6.2 Web Server Application

We evaluate the JOS web server [11] in KylinX, which adopts multithreading for multiple connections. After the main thread accepts an incoming connection, the web server creates a worker thread to parse the header, reads the file, and sends the contents back to the client. We use the Weighttp benchmark that supports a small fraction of the HTTP protocol (but enough for our web server) to measure the web server performance. Similar to the evaluation of Redis server, we test the web server by running multiple Weighttp [8] clients on one machine, each continuously sending `GET` requests to the web server.

We evaluate the throughput as a function of the number of concurrent clients, and compare it with the web servers running on MiniOS and Ubuntu, respectively. The result is depicted in Fig. 9, where the KylinX web server achieves higher throughput than the MiniOS web server since it provides higher sequential performance. Both KylinX and MiniOS web servers are slower than the Ubuntu web server, because the asynchronous `select` is inefficiently scheduled with the netfront driver of MiniOS [27].

5 Related Work

KylinX is related to static Unikernel appliances [56, 27], reduced VMs [19, 48, 49], containers [66, 9, 15], and picoprocess [38, 62, 32, 54, 67, 33].

5.1 Unikernel & Reduced VMs

KylinX is an extension of Unikernel [56] and is implemented on top of MiniOS [27]. Unikernel OSs include Mirage [56], Jitsu [55], Unikraft [18], etc. For example, Jitsu [55] leverages Mirage [56] to design a power-efficient and responsive platform for hosting cloud services in the edge networks. LightVM [58] leverages Unikernel on Xen to achieve fast booting.

MiniOS [27] designs and implements a C-style Unikernel LibOS that runs as a para-virtualized guest OS within a Xen domain. MiniOS has better backward compatibility than Mirage and supports single-process applications written in C. However, the original MiniOS statically seals an appliance and suffers from similar problems with other static Unikernels.

The difference between KylinX and static Unikernels (like Mirage [56], MiniOS [27], and EbbRT [65]) lies in the pVM abstraction which explicitly takes the hypervisor as an OS and supports process-style operations like pVM fork/IpC and dynamic library mapping. Mapping restrictions (§3.3.4) make KylinX introduce as little vulnerability as possible and have no larger TCB than Mirage/MiniOS [56, 55]. KylinX supports source code (C) compatibility instead of using a type-safe language to rewrite the entire software stack [56].

Recent research [19, 49, 48] tries to improve the hypervisor-based type-1 VMs to achieve smaller memory footprint, shorter boot times, and higher execution performance. Tiny Core Linux [19] trims an existing Linux distribution down as much as possible to reduce the overhead of the guest. OS^v [49] implements a new guest OS for running a single application on a VM, resolving libc function calls to its kernel that adopts optimization techniques such as the spinlock-free mutex [70] and the net-channel networking stack [46]. RumpKernel [48] reduces the VMs by implementing a optimized guest OS. Different from KylinX, these general-purpose LibOS designs consist of unnecessary features for a target application leading to larger attack surface. They cannot support the multi-process abstraction. Besides, KylinX's pVM fork is much faster than replication-based VM_fork in SnowFlock [50].

5.2 Containers

Containers use OS-level virtualization [66] and leverage kernel features to package and isolate processes, instead of relying on the hypervisors. In return they do not need to trap syscalls or emulate hardware, and could run as normal OS processes. For example, Linux Containers (LXC) [9] and Docker [15] create containers by using a number of Linux kernel features (such as *namespaces* and *cgroups*) to package resource and run container-based processes.

Containers require to use the same host OS API [49], and thus expose hundreds of system calls and enlarging the attack surface of the host. Therefore, although LXC and Docker containers are usually more efficient than traditional VMs, they provide less security guarantees since attackers may compromise processes running inside containers.

5.3 Picoprocess

A picoprocess is essentially a container which implements a LibOS between the host OS and the guest, mapping high-level guest API onto a small interface. The original picoprocess designs (Xax [38] and Embassies [43]) only permit a tiny syscall API, which can be small enough to be convincingly (even verifiably) isolated. Howell et al. show how to support a small subset of single-process applications on top of a minimal picoprocess interface [44], by providing a POSIX emulation layer and binding existing programs.

Recent studies relax the static and rigid picoprocess isolation model. For example, Drawbridge [62] is a Windows translation of the Xax [38] picoprocess, and creates a picoprocess LibOS which supports rich desktop applications. Graphene [67] broadens the LibOS paradigm by supporting multi-process API in a family (sandbox) of picoprocesses (using message passing). Bascule [32] allows OS-independent extensions to be attached safely and efficiently at runtime. Tardigrade [54] uses picoprocesses to easily construct fault-tolerant services. The success of these relaxations on picoprocess inspires our dynamic KylinX extension to Unikernel.

Containers and picoprocesses often have a large TCB since the LibOSs contain unused features. In contrast, KylinX and other Unikernels leverage the hypervisor's virtual hardware abstraction to simplify their implementation, and follow the *minimalism* philosophy [36] to link an application only against requisite libraries to improve not only efficiency but also security.

Dune [34] leverages Intel VT-x [69] to provide a process (rather than a machine) abstraction to isolate processes and access privileged hardware features. IX [35] incorporates virtual devices into the Dune process model and achieves high throughput and low latency for networked systems. lwCs [53] provides independent units of protection, privilege, and execution state within a process.

Compared to these techniques, KylinX runs directly on Xen (a type-1 hypervisor), which naturally provides strong isolation and enables KylinX to focus on the flexibility and efficiency issues.

6 Conclusion

The tension between strong isolation and rich features has been long lived in the literature of cloud virtualization. This paper exploits the new design space and proposes the pVM abstraction by adding two new features (dynamic page and library mapping) to the highly-specialized static Unikernel. The simplified virtualization architecture (KylinX) takes the hypervisor as an OS and safely supports flexible process-style operations such as pVM fork and inter-pVM communication, runtime update, and fast recycling.

In the future, we will improve security through modularization [27], disaggregation [60], and SGX enclaves [33, 28, 45, 68]. We will improve the performance of KylinX by adopting more efficient runtime like MUSL [23], and adapt KylinX to the MultiLibOS model [65] which allows spanning pVMs onto multiple machines. Currently, the pVM recycling mechanism is still tentative and conditional: it can only checkpoint an empty domain; the recycled pVM cannot communicate with other pVMs using event channels or shared memory; the application can only be in the form of a self-contained shared library that does not need to load/link other shared libraries; and there are still no safeguards inspecting potential security threats between the new and old pVMs after recycling. We will address these shortcomings in our future work.

7 Acknowledgements

This work was supported by the National Basic Research Program of China (2014CB340303) and the National Natural Science Foundation of China (61772541). We thank Dr. Ziyang Li, Qiao Zhou, and the anonymous reviewers for their help in improving this paper. This work was performed when the first author was visiting the NetOS group of Computer Lab at University of Cambridge, and we thank Professor Anil Madhavapeddy, Ripduman Sohan and Hao Liu for the discussion.

References

[1] http://erlangonxen.org/.

[2] http://luajit.org/.

[3] http://man7.org/linux/manpages/man2/syscalls.2.html.

[4] http://savannah.nongnu.org/projects/lwip/.

[5] https://aws.amazon.com/ec2/.

[6] https://aws.amazon.com/lambda/.

[7] https://azure.microsoft.com/en-us/services/container-service/.

[8] https://github.com/lighttpd/weighttp/.

[9] https://linuxcontainers.org/.

[10] https://ocaml.org/.

[11] https://pdos.csail.mit.edu/6.828/2014/labs/lab6/.

[12] https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf.

[13] https://redis.io/.

[14] https://wiki.xenproject.org/wiki/Mini-OS.

[15] https://www.docker.com/.

[16] https://www.gnu.org/software/libc/.

[17] https://www.openbsd.org/papers/eurobsdcon2014_securelazy/slide003a.html.

[18] https://www.xenproject.org/developers/teams/unikraft.html.

[19] http://tinycorelinux.net/.

[20] http://www.gamesparks.com/.

[21] http://www.java.com/.

[22] http://www.linux-kvm.org/.

[23] http://www.musl-libc.org.

[24] http://www.sas.com/en_us/home.html.

[25] http://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/.

[26] AL-KISWANY, S., SUBHRAVETI, D., SARKAR, P., AND RIPEANU, M. Vmflock: virtual machine co-migration for the cloud. In *Proceedings of the 20th international symposium on High performance distributed computing* (2011), ACM, pp. 159–170.

[27] ANDERSON, M. J., MOFFIE, M., AND DALTON, C. I. Towards trustworthy virtualisation environments: Xen library os security service infrastructure. *HP Tech Reort* (2007), 88–111.

[28] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., OKEEFFE, D., STILLWELL, M. L., ET AL. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, GA* (2016).

[29] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), ACM, pp. 187–198.

[30] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 164–177.

[31] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 29–44.

[32] BAUMANN, A., LEE, D., FONSECA, P., GLENDENNING, L., LORCH, J. R., BOND, B., OLINSKY, R., AND HUNT, G. C. Composing os extensions safely and efficiently with bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 239–252.

[33] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[34] BELAY, A., BITTAU, A., MASHTIZADEH, A., TEREI, D., MAZIÈRES, D., AND KOZYRAKIS, C. Dune: Safe user-level access to privileged cpu features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012), pp. 335–348.

[35] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 49–65.

[36] BROCKMANN, R. J. The why, where and how of minimalism. In *ACM SIGDOC Asterisk Journal of Computer Documentation* (1990), vol. 14, ACM, pp. 111–119.

[37] CHEN, H., YU, J., CHEN, R., ZANG, B., AND YEW, P.-C. Polus: A powerful live updating system. In *Proceedings of the 29th international conference on Software Engineering* (2007), IEEE Computer Society, pp. 271–281.

[38] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *OSDI* (2008), vol. 8, pp. 339–354.

[39] ENGLER, D. R., KAASHOEK, M. F., ET AL. Exokernel: An operating system architecture for application-level resource management. In *Fifteenth ACM Symposium on Operating Systems Principles* (1995), ACM.

[40] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM 53*, 10 (2010), 85–93.

[41] HAYDEN, C. M., SMITH, E. K., DENCHEV, M., HICKS, M., AND FOSTER, J. S. Kitsune: Efficient, general-purpose dynamic software updating for c. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 249–264.

[42] HEO, J., ZHU, X., PADALA, P., AND WANG, Z. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on* (2009), IEEE, pp. 630–637.

[43] HOWELL, J., PARNO, B., AND DOUCEUR, J. R. Embassies: Radically refactoring the web. In *NSDI* (2013), pp. 529–545.

[44] HOWELL, J., PARNO, B., AND DOUCEUR, J. R. How to run posix apps in a minimal picoprocess. In *USENIX Annual Technical Conference* (2013), pp. 321–332.

[45] HUNT, T., ZHU, Z., XU, Y., PETER, S., AND WITCHEL, E. Ryoan: a distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation* (2016), USENIX Association, pp. 533–549.

[46] Jacobson, V., and Felderman, B. Speeding up networking. In *Ottawa Linux Symposium (July 2006)* (2006).

[47] Kamp, P.-H., and Watson, R. N. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference* (2000), vol. 43, p. 116.

[48] Kantee, A., and Cormack, J. Rump kernels: no os? no problems! *The magazine of USENIX & SAGE 39*, 5 (2014), 11–17.

[49] Kivity, A., Laor, D., Costa, G., Enberg, P., HarEl, N., Marti, D., and Zolotarov, V. Osv: optimizing the operating system for virtual machines. In *2014 usenix annual technical conference (usenix atc 14)* (2014), pp. 61–72.

[50] Lagar-Cavilla, H. A., Whitney, J. A., Scannell, A. M., Patchin, P., Rumble, S. M., De Lara, E., Brudno, M., and Satyanarayanan, M. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems* (2009), ACM, pp. 1–12.

[51] Lee, I. Dymos: a dynamic modification system.

[52] Li, H., Zhang, Y., Zhang, Z., Liu, S., Li, D., Liu, X., and Peng, Y. Parix: Speculative partial writes in erasure-coded systems. In *USENIX Annual Technical Conference* (2017), USENIX Association, pp. 581–587.

[53] Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., and Druschel, P. Light-weight contexts: An os abstraction for safety and performance. In *OSDI* (2016), USENIX, pp. 49–64.

[54] Lorch, J. R., Baumann, A., Glendenning, L., Meyer, D. T., and Warfield, A. Tardigrade: Leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015* (2015), pp. 575–588.

[55] Madhavapeddy, A., Leonard, T., Skjegstad, M., Gazagnaire, T., Sheets, D., Scott, D., Mortier, R., Chaudhry, A., Singh, B., Ludlam, J., et al. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked System Design and Implementation* (2015).

[56] Madhavapeddy, A., Mortier, R., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. In *ACM ASPLOS* (2013), ACM, pp. 461–472.

[57] Madhavapeddy, A., Mortier, R., Sohan, R., Gazagnaire, T., Hand, S., Deegan, T., McAuley, D., and Crowcroft, J. Turning down the lamp: software specialisation for the cloud. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, HotCloud* (2010), vol. 10, pp. 11–11.

[58] Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., and Huici, F. My vm is lighter (and safer) than your container.

[59] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), USENIX Association, pp. 459–473.

[60] Murray, D. G., Milos, G., and Hand, S. Improving xen security through disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (2008), ACM, pp. 151–160.

[61] Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. *Practical dynamic software updating for C*, vol. 41. ACM, 2006.

[62] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R., and Hunt, G. C. Rethinking the library os from the top down. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011), ACM, pp. 291–304.

[63] Ramaswamy, A., Bratus, S., Smith, S. W., and Locasto, M. E. Katana: A hot patching framework for elf executables. In *Availability, Reliability, and Security, 2010. ARES'10 International Conference on* (2010), IEEE, pp. 507–512.

[64] Rutkowska, J., and Wojtczuk, R. Qubes os architecture. *Invisible Things Lab Tech Rep 54* (2010).

[65] Schatzberg, D., Cadden, J., Dong, H., Krieger, O., and Appavoo, J. Ebbrt: A framework for building per-application library operating systems. In *OSDI* (2016), USENIX, pp. 671–688.

[66] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 275–287.

[67] Tsai, C.-C., Arora, K. S., Bandi, N., Jain, B., Jannen, W., John, J., Kalodner, H. A., Kulkarni, V., Oliveira, D., and Porter, D. E. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 9.

[68] Tsai, C.-C., Porter, D. E., and Vij, M. Graphene-sgx: A practical library os for unmodified applications on sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC)* (2017).

[69] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F., Anderson, A. V., Bennett, S. M., Kägi, A., Leung, F. H., and Smith, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[70] Uhlig, V., LeVasseur, J., Skoglund, E., and Dannowski, U. Towards scalable multiprocessor virtual machines. In *Virtual Machine Research and Technology Symposium* (2004), pp. 43–56.

[71] Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K., and Wood, T. Flurries: Countless fine-grained nfs for flexible per-flow customization. In *CoNEXT* (2016), ACM, pp. 3–17.

Virtualizing Energy Storage Management Using RAIBA

Tzi-cker Chiueh[1] Mao-Cheng Huang Kai-Cheung Juang Shih-Hao Liang Welkin Ling

Industrial Technology Research Institute

Abstract

Because of the intermittent nature of renewable energy-based electricity generation, a key building block for a sustainable renewable energy-based electricity infrastructure is cost-effective energy storage management, which is largely determined by the cost of electric batteries. Despite substantial technological advances in recent years, batteries used in consumer devices and electric vehicles are still too expensive to be feasible for large-scale deployment. One promising way to reduce the battery cost of an energy storage system is to leverage retired batteries from electric vehicles. However, because the charging/discharging characteristics of retired batteries tend to vary widely from one another, putting these heterogeneous batteries into the same module or energy storage system pose significant safety risks and efficiency challenges. This paper presents the design, implementation and evaluation of a *reconfigurable* battery array system called *RAIBA* that is designed to address the heterogeneity issue in retired battery-based energy storage systems by allowing the inter-battery connectivity to be reconfigurable at run time. In addition, *RAIBA* also enables virtualization of the electrical energy resources in a battery array in the same way as how computing, storage and network resources are virtualized. Empirical measurements on a fully operational *RAIBA* prototype demonstrate that it can effectively increase the discharge service time by more than 80% under a set of real-world electric load traces.

1 Introduction

In light of the global warming and resulting climate change effects triggered by carbon-based fossil energy sources, more and more counties are charging ahead to build an energy infrastructure in which renewable energy plays a major role. Because most important renewable energy sources, such as sun, wind and tidal wave, generate electricity in a way dictated by weather conditions, this intermittent nature renders them unfit as a base load electric energy source, unless they are supplemented by an energy storage system that could bridge the time gaps between energy production and energy consumption. Therefore, for renewable energy to become a significant element of the future clean energy infrastructure, cost-effective energy storage management is a key factor. Most grid-scale energy storage systems today are pumped hydroelectric energy storage, which stores energy in the form of gravitational potential energy of water, which is pumped by off-peak low-cost electricity from a lower-elevation reservoir to one with a higher elevation. However, as electric vehicles become more and more prevalent, the electric batteries they use will increasingly become an important part of the energy storage element of the renewable energy infrastructure.

Despite substantial technological advances in recent years, batteries used in consumer devices and electric vehicles are still too expensive to be feasible for large-scale deployment. In 2017, the LCOE (levelized cost of electricity) of Lithium-based battery [18] is about $0.4 USD per kWh discharge, but US DOE's LCOE target [8] for a cost-effective renewable energy storage is $0.1 USD per kWh discharge. A promising way to facilitate the reduction of the battery cost of an energy storage system is to leverage retired batteries from electric vehicles, as has already been done by several electric or hybrid vehicle manufacturers, such as Tesla, Nissan, Toyota, and BMW. The reason that batteries retired from electric vehicles are still usable as energy storage is because the residual capacity of retired batteries is generally between 70% to 80% of their original full capacity. However, the charging/discharging characteristics of these retired batteries may deviate substantially from those when they were in mint condition.

Conventional battery energy storage systems assume that the charging/discharging characteristics of constituent batteries are homogeneous, as it greatly reduces

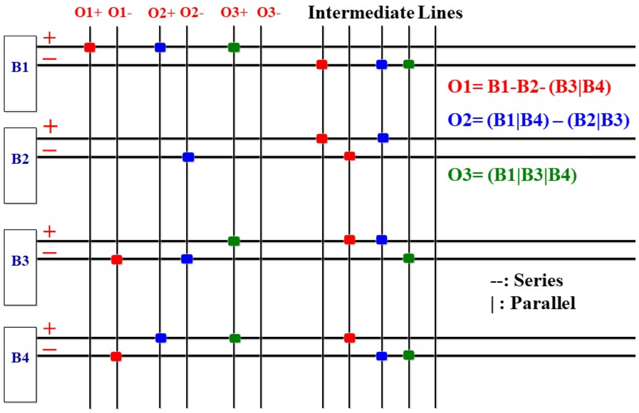


Figure 1: *The most general form of the proposed* RAIBA *architecture, in which each battery could be attached or detached from the interconnect, and any battery could be connected to any other battery in series or in parallel. Intersecting signal lines are connected when they are joined by a dot.*

energy utilization inefficiency. For example, consider a set of batteries that are connected in series, and the capacity of one of them is significantly smaller than the others. In this case, the effective charging/discharging capacity of the entire battery series is dictated by the weakest battery, and the additional capacity of all the other stronger batteries is essentially left wasted. Because retired batteries come from a wide variety of sources, their charging/discharging characteristics are bound to vary widely from one another, and putting these heterogeneous batteries into the same module or energy storage system pose safety risks and utilization efficiency challenges.

In this work, we propose a *Reconfigurable Array of Inexpensive Battery Architecture* (*RAIBA*) [12, 7, 6] to address the heterogeneity issue inherent in energy storage systems that are built from retired batteries. Tesla pioneered the idea of applying commodity batteries used in 3C consumer devices, i.e., 18650 lithium batteries, to building large-scale battery arrays used in electric vehicles. For example, the number of 18650 batteries used in Tesla Model S's battery array is more than 7,000. These battery arrays are provisioned with a certain amount of redundancy to cope with potential battery failures, but when a battery fails, the entire module containing it is impacted because the inter-battery connectivity is fixed. To more effectively minimize the impacts of battery failures and degradations, we propose that a battery array be *reconfigurable* so as to work around failed or degraded batteries at run time.

The most general form of *RAIBA* is shown in Figure 1, which, via software control, controls whether each battery is attached to or detached from the array's interconnect, and how the batteries are connected to one another at run time. The three outputs, O1, O2 and O3, shown in red, blue and green in the figure, represent the outputs of this 4-battery array at three different points in time. The first output (O1) is driven by a series connection among Battery 1 (B1), Battery 2 (B2), and a parallel connection between Battery 3 (B3) and Battery 4 (B4), whereas the second output (O2) corresponds to a series connection between a parallel connection between Battery 1 (B1) and Battery 4 (B4), and another parallel connection between Battery 2 (B2) and Battery 3 (B3). The third output (O3), to which Battery 2 (B2) does not contribute, is simply a parallel connection among B1, B3 and B4. The dynamic reconfigurability afforded by *RAIBA* allows the inter-battery connectivity to be tailored to a given electrical load so as to make the best of available battery resources and minimize unnecessary energy loss, as illustrated by the following three use cases:

- For a series-connected battery array, when the capacity of the weakest battery is exhausted during a discharge operation, *RAIBA* could temporarily put it aside to prevent it from blocking the entire array, and then continues the discharge operation by making the best of the additional capacities of other stronger batteries.
- For a parallel-connected battery array, when the capacity of the weakest battery becomes significantly smaller than that of the others during a discharge operation, *RAIBA* could temporarily put it aside to prevent unwanted inter-battery capacity balancing, which consumes energy, and then continues the discharge operation by making the best of the additional capacities of other stronger batteries.
- Given an electric load request, *RAIBA* makes it possible to use a proper subset of batteries to provide an aggregate voltage and current level which *barely* exceeds those of the request, with their differences and thus the associated down-conversion losses being reduced to the minimum .

The reconfigurability of *RAIBA* opens up myriad battery resource management flexibilities that are not previously possible, and in particular enables virtualization of a physical battery pool in a way similar to how computing and networking resources are virtualized. Given an electric load request $< V, I >$, *RAIBA* dynamically configures a *virtual battery* best fit to satisfy the request by first selecting a proper subset of qualified batteries, and then connecting them in the most appropriate way, so that the virtual battery's aggregate voltage level exceeds V, its aggregate current exceeds I, and the incurred energy loss due to conversion and balancing is minimized.

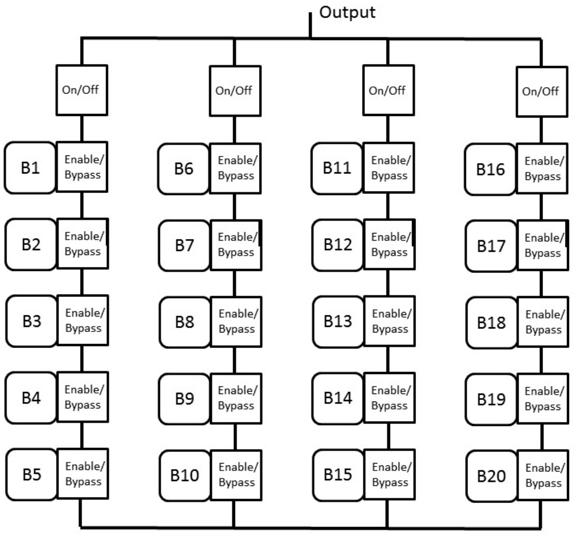


Figure 2: *The system architecture of* RAIBA-1 *consists of N columns connected in parallel, each of which contains M batteries connected in series, with each battery's connectivity to the interconnect via an* enable/bypass *switch.*

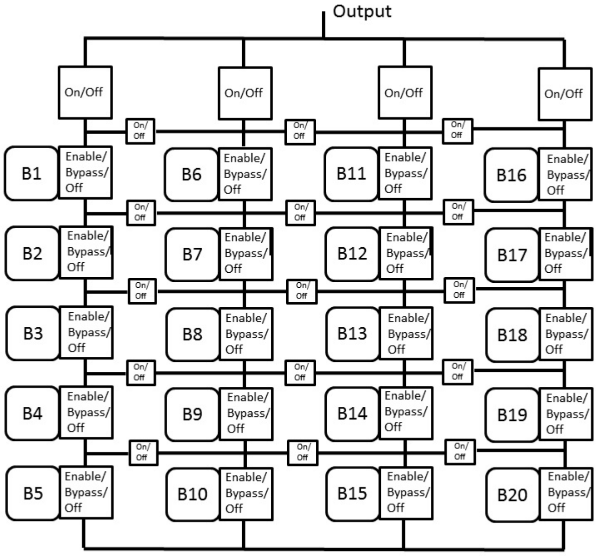


Figure 3: *The system architecture of* RAIBA-2 *is a refinement of* RAIBA-1, *with each* enable/bypass *switch replaced with an* enable/bypass/off *switch, and horizontal inter-column connectivity controlled by* on/off *switches.*

2 RAIBA Levels

The inter-battery interconnect of a *RAIBA* system shown in Figure 1 is deceivingly similar to that of a programmable logic array (PLA) digital logic circuit [14], with a series connection corresponding to an AND operation and a parallel connection corresponding to an OR operation. But the analogy quickly breaks down because of the following two technical challenges:

- The inter-battery interconnect is an energy delivery network that is tasked with carrying much larger electric energy than typical digital logic circuits, and thus require advanced power electronics circuit design techniques to support complex connectivity patterns while guaranteeing operational safety.
- To render inter-battery connectivity reconfiguration completely seamless to an electric load served by a *RAIBA* system, *RAIBA* requires *on-line reconfigurability*, which means the entire system remains functional across each reconfiguration operation and the electric energy pattern delivered to the load before a reconfiguration operation is very close to that after the reconfiguration operation.

Taking into account the significant added circuit complexities involved in supporting dynamic reconfiguration of inter-battery connectivity, we design three RAIBA levels that offer increasing reconfiguration flexibility but also impose growing circuit design challenges.

A *RAIBA-1* array, as shown in Figure 2, consists of *N* columns connected in parallel, each of which contains *M* batteries connected in series. At any point in time, each battery is either part of or insulated from a certain column through an *enable/bypass* switch. When the switch is *enable*, the battery participates in the series connectivity of the column; when the switch is *bypass*, the battery takes itself off the series connectivity of the column. Moreover, above all batteries in each column is an *on/off* switch that allows a column to participate in or sit out of the overall inter-column parallel connection. Each battery in a RAIBA-1 system could only be connected in series with other batteries in the same column.

A *RAIBA-2* array, as shown in Figure 3, is basically a *RAIBA-1* array augmented with row-wise inter-column connectivity controllable by *on/off* switches. In addition, each battery is equipped with a *enable/bypass/off* switch that allows a battery to be part of or insulated from its column, or to disconnect the batteries above it in the same column from those below it. That is, when a battery's switch to the battery array's interconnect is off, the battery breaks off the series connectivity of the column to which it belongs. The switchable inter-column interconnects run horizontally and provide additional connectivity flexibility of allowing a battery in the *i*-th column to be connected in series with another battery in the *j*-th column, or be connected in parallel with another battery in the *j*-th column without bypassing all other batteries in the *i*-th and *j*-th column. For example, if the horizontal on/off switches above and below B4 and B9 are turned on, B4 and B9 are effectively connected in parallel; if B3 is off and the on/off switch above B4 and B9 is turned on, B4 and B8 are effectively connected in series.

While *RAIBA-1* and *RAIBA-2* are designed to support a single electric load at a time, *RAIBA-3*, as shown in Figure 1, is able to support multiple electric loads simultaneously. In addition, a *RAIBA-3* array allows any battery to be connected in series or in parallel with any other battery in the array. This generality provides much more room for battery resource optimization, but requires each battery to be equipped with approximately as many *on/off* switches as the sum of the numbers of the output lines and intermediate lines, a significant increase hardware implementation complexity.

Although increasing RAIBA levels offer more dynamic reconfigurability, whether the additional hardware complexity associated with higher RAIBA levels is worth the potential gains due to the additional flexibility they provide is an open question. For the rest of this paper, we will only focus on the *RAIBA-1* architecture.

3 Hardware Support

The key building block of the *RAIBA-1* architecture is the *enable/bypass* switch, which either enables a battery to participate in its column, i.e., connecting it in series with the other batteries, or bypasses a battery from its column, i.e., insulating it from the other batteries. While conceptually straightforward, two operational requirements render its design technically challenging:

- The transition between the enable mode and the bypass mode should be as short as possible so as to minimize the energy consumed by each transition.
- The electric current flowing through a *enable/bypass* switch should remain constant during each transition between the enable mode and the bypass mode, so as to minimize the disruption to the electric loads being served.

We have designed and implemented an analog ASIC for the enable/bypass switch, whose architecture and circuit layout are shown in Figure 4. Rather than with an individual battery, this ASIC is designed to work with a battery module, which in turn consists of multiple batteries, in this example, 5 batteries connected in series. An enable/bypass switch includes two on/off switches, S1 and S2, a *battery monitor*, which keeps track of the temperature (Tsen), voltage (Vsen) and current (Isen) of each of batteries in the module, and a *Mux controller*, which controls how S1 and S2 are turned on and off. When S1 is on and S2 is off, the enable/bypass switch bypasses the battery module and the voltage drop across the enable/bypass switch is zero. When S1 is off and S2 is on, the enable/bypass switch enables the battery module and so the voltage drop across the enable/bypass switch is the voltage difference between the two ends of the battery module. When S1 and S2 are both off, the enable/bypass switch cuts off the entire column to which it is connected. When S1 and S2 are both on, the effective circuit becomes a short circuit, and I2 may grow to a dangerously large level. Therefore, this mode of operation is strictly prohibited.

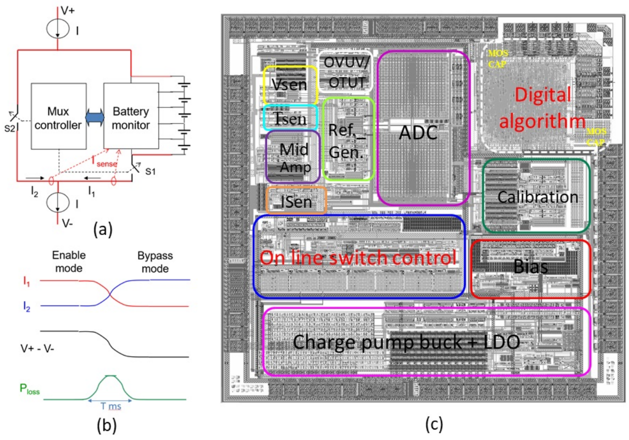


Figure 4: *The (a) circuit architecture and (b) physical layout of the ASIC implementing the* enable/bypass *switch required by the* RAIBA-1 *architecture. (b) shows its ideal transient electric circuit behavior during a transition between the enable and the bypass mode.*

When an enable/bypass switch goes from the enable (bypass) mode to the bypass (enable) mode, the current running through S1, I1, is decreasing (increasing), but the current running through S2, I2, is increasing (decreasing). During a transition between the enable mode and the bypass mode, the *Mux controller* constantly measures I1 and I2, and applies a feedback control mechanism to dynamically tuning the degree to which S1 and S2 are on so that the sum of I1 and I2 remains constant throughput the transition.

To decrease the power consumption incurred by each transition (P_{loss}), the amount of time during which neither S1 nor S2 is off should be minimized, because P_{loss} is equal to the product of the voltage drop and the sum of I1 and I2 during this period, as shown in the lower left of Figure 4. Minimizing the length of the transition period conflicts with the goal of keeping the sum of I1 and I2 constant during the transition period, because, intuitively, it is easier to slowly tune S1 and S2 to keep the total current constant than to try to do so quickly.

When a battery module is enabled by an enable/bypass switch, it may seem that S1 could incur additional energy consumption due to its on-resistance, which is typically very small. However, even for a non-*RAIBA* system, each battery module is typically paired with an on/off switch in order to protect the module from being damaged by unexpected charging currents. This protection on/off switch is no different from S1.

4 Configuration Control Algorithm

Because of its dynamic configurability, *RAIBA* is able to apply a different configuration to each electric load request. Given an electric load $< V, I >$, *RAIBA*'s configuration control algorithm computes a configuration that best serves this load using the following optimization criteria:

- The configuration's delivered current and voltage level exceed I and V, respectively.
- The difference between the configuration's delivered power and $V * I$ is minimized.
- The residual capacities of the batteries in the array are as equalized as possible.

Because every configuration change itself incurs energy loss, *RAIBA* keeps on using its current configuration to satisfy a new electric load request until either the current configuration cannot satisfy the load's $< I, V >$ requirement or the batteries in the current configuration is seriously imbalanced.

The design of *RAIBA*'s configuration control algorithm aims to maximize the energy output of each charge/discharge cycle and the total number of charge/discharge cycles. To squeeze out every bit of the energy accumulated in a charge cycle, it is essential that the left-over battery capacity at the end of a charge cycle be reduced to the minimum. The most likely scenario of squandered battery capacity occurs when one of the batteries in a series-connected battery chain exhausts its capacity and the remaining capacities of the other batteries in the chain are forced to be laid to waste. To avoid this, one should balance the residual capacities of a *RAIBA* array as much as possible. The key to maximizing a battery array's total number of charge cycles is to use each battery in it as gently as possible. Towards this end, when serving an electric load, it is desirable to involve as many batteries and thus draw as little electric current from each battery as possible.

Even though an *enable/bypass* switch in a *RAIBA* system is associated with a battery module, to simplify the exposition below we will assume that each battery modules consists of a single battery. The configuration control algorithm (CCA) used in the current *RAIBA* prototype is shown in Figure 5. Designed with in mind the above optimization objectives, CCA first identifies all possible configurations in an NxM battery array that meet the requirements of the given electric load request, and then picks the one that best balances the residual capacities of all the batteries in the array. Instead of trying out all possible battery combinations, CCA takes a greedy approach by processing each column independently, and within each column, considering only battery combinations that consist of top K batteries in the column's battery list sorted according to their residual capacity and whose aggregate voltage level lies between V and $\alpha * V$. A column is disqualified if the aggregate voltage level of all M batteries in it is below V. The α parameter bounds the search scope and prevents CCA from examining "over-provisioned" configurations. If no satisfactory configuration could be identified for a given α value, CCA increases α and repeats the algorithmic process in Figure 5 again.

```
Input: $< I, V >$ of an electric load request, and the
      residual capacity and voltage level of each
      battery in an NxM array;
for (each of the N columns) {
    Sort the batteries in the column according to
    their residual capacity into a list;

    Traverse the list in the sorted order, find all
    possible battery combinations whose
    accumulated voltage level is between $V$ and $V * \alpha$,
    and place the resulting combinations into a set;

    Disqualify the column if its set is empty;
}

Form a candidate configuration by picking one
battery combination from each qualified column's
set, and put all candidate configurations into a
list, CCL;

for (each candidate configuration in CCL) {
    Simulate the candidate configuration for one
    time step according to a battery model derived
    from dynamic measurements;

    Compute the candidate configuration's
    eventual output voltage level, $V_{out}$, and
    switching cost, $Cost_{switch}$;

    Derive the residual capacity of all
    participating batteries one time step later,
    and compute the standard deviation of the
    residual capacities of all batteries, $STD_c$;

    Disqualify the candidate configuration if
    the current going through any participating
    column is negative or its total power output
    is less than $V * I$;
}

Select the qualified candidate configuration that
minimizes $\beta * \frac{V_{out} - V}{V_{out}}$ + $\gamma * STD_c$ + $\delta * Cost_{switch}$;
```

Figure 5: RAIBA*'s configuration control algorithm, which aims to balance the aggregate voltage levels of participating columns and the residual capacities of all the batteries*

When the aggregate voltage level of one column of a candidate configuration is significantly lower than those of the other columns, other columns may charge the weaker column to bring its voltage level up to par with others, in which case the electric current going through the weaker column is *negative*, or opposite in direction to the electric current requirement (I) of the electric load request. Because the overhead incurred by such inter-column charging represents an unnecessary energy loss, CCA disqualifies all candidate configurations that lead to inter-column charging from consideration.

To maximize the life time of an array's batteries, CCA spreads an electric load over as many columns as possible by considering only those configurations that include all the qualified columns. That is, if L columns in an N-column array are qualified, CCA considers only configurations that consists of all L columns, but not those consisting of a proper subset of these L columns. It is conceivable that configurations using fewer than L columns could lead to lower STD_c or V_{out} or both, but including them into consideration would significantly enlarge the search space.

CCA takes into account the following three factors when selecting the best among the candidate configurations. First, to reduce the amount of wasted battery capacity at the end of a charge cycle, CCA strives to balance the residual capacity of an array's batteries, STD_c, by minimizing the standard deviation of their residual capacity after a time period. Second, to reduce the energy loss due to the inverter, which down-converts a candidate configuration's eventual output voltage (V_{out}) to V, CCA also minimizes the difference between V_{out} and V. Finally, because each switching of an *enable/bypass* switch also incurs an energy loss, it is desirable to pick a configuration that is as close to the current configuration as possible. For this, CCA computes the switching energy cost of each candidate configuration, $Cost_{switch}$. Because these factors may conflict with one another, CCA uses three empirically determined parameters, β, γ and δ, to adjust their relative importance or weight.

Given a candidate configuration that comprises L columns, each of which consists of a variable number of batteries, to execute the above algorithm, CCA needs to compute the eventual output voltage V_{out}, the current going through each of the columns, and the residual capacity of each participating battery after a time period. Because of the non-linear discharging characteristics of modern batteries, CCA resorts to a simulation approach to deriving the equivalent electrical circuit behavior of each participating battery. Instead of pre-calibrating each battery in advance, CCA adopts a *trace-based* strategy to build up a simulation model for each battery by periodically measuring the instantaneous *discharge current*, *voltage level* and *used capacity* when it is discharged, generating a sampled version of its discharge characteristic curves (DCC) [17, 19], which describes how a battery's voltage level evolves with its used (not residual) capacity at a discharge current, and then applying linear interpolation to approximating those points on the DCCs that do not have measured values. For example, Figure 6 shows the measured DCCs for a 400-cycle Panasonic NCR 18650B battery for four different discharge currents, 0.31A, 1.55A, 3.1A, and 4.65A, with measurements taken in the temperature range between 25 and 35 degrees Celsius. Note that there is a distinct linear and thus easier to predict range for the DCC corresponding to a particular discharge current. The smaller the discharge current, the larger the corresponding DCC's linear range. Also, suppose the cut-off voltage level is 3V, then the total usable capacity of this battery is around 2Ah when it is discharged at 4.65A, but is about 2.95Ah when it is discharged at 0.31A. This example illustrates that discharging batteries as gently as possible not only lengthens their total lifetime, but also increases their per-charge usable capacity.

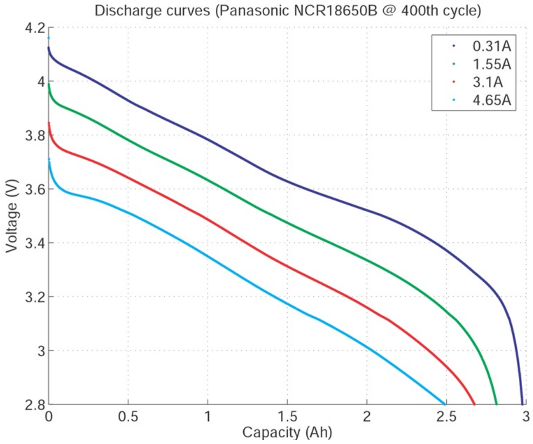


Figure 6: *The discharge characteristic curves for a Panasonic NCR 18650B battery that has been charged 400 times for four different discharge currents, 0.31A, 1.55A, 3.1A, and 4.65A. Measurements were taken in the temperature range between 25 and 35 degrees Celsius.*

CCA models a battery as a voltage source connected in series with an internal resistance. For an L-column candidate configuration, CCA assumes the initial current going through each column and thus each battery in the array is $\frac{I}{L}$. To compute the internal resistance of a battery with a used capacity X and a discharge current of $\frac{I}{L}$, CCA first identifies the two DCC measurements (<discharge current, voltage level, used capacity>) that are closest to $< \frac{I}{L},, X >$, say $< I_a, V_a, UC_a >$ and $< I_b, V_b, UC_b >$, and then approximates its internal resistance as $\frac{\triangle V}{\triangle I} = \frac{V_b - V_a}{I_a - I_b}$. For example, to calculate the internal resistance of the Panasonic battery whose DCC is shown in Figure 6 when

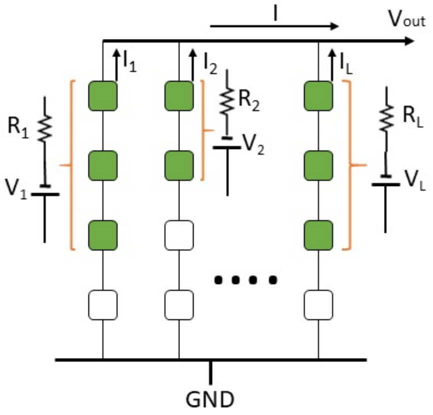


Figure 7: *The equivalent circuit of an L-column candidate configuration, which selects the first 3 batteries of the first column, the first 2 batteries of the second column,... and the first 3 batteries of the L-th column. Each column's effective resistance is the sum of the internal resistances of the participating batteries in that column.*

its used battery is 2Ah and discharge current is 3.6A, we identify the closest measurements $< 3.0V, 4.65A, 2Ah >$ and $< 3.18V, 3.1A, 2Ah >$, and then compute its internal resistance as $\frac{3.18-3.0}{4.65-3.1} = 0.12$ Ohm.

With the internal resistance of every participating battery, CCA computes the aggregate voltage level (V_i)and aggregate internal resistance R_i for each column in the candidate configuration, represents the L-column candidate configuration as an equivalent circuit as shown in Figure 7, and then solves the corresponding linear system of equations as follows, to derive V_{out} and the actual current going through each column, I_i:

$$\begin{aligned} V_1 - I_1 * R_1 &= V_{out} \\ V_2 - I_2 * R_2 &= V_{out} \\ V_3 - I_3 * R_3 &= V_{out} \\ &\ldots\ldots\ldots \\ V_L - I_L * R_L &= V_{out} \\ I_1 + I_2 + I_3 + \ldots I_L &= I \end{aligned}$$

If V_{out} is larger than some V_j, then the corresponding current I_j must be negative, the corresponding (j-th) column is disqualified, and the associated candidate configuration is considered unusable. Once the current going through each column of a candidate configuration is known, CCA computes, for each participating battery, the amount of charge that will be discharged within a fixed time period by multiplying the discharge current with the period's length, subtracts the multiplication result from the battery's residual capacity, and finally derives the standard deviation of the residual capacities of all batteries in the configuration, STD_c.

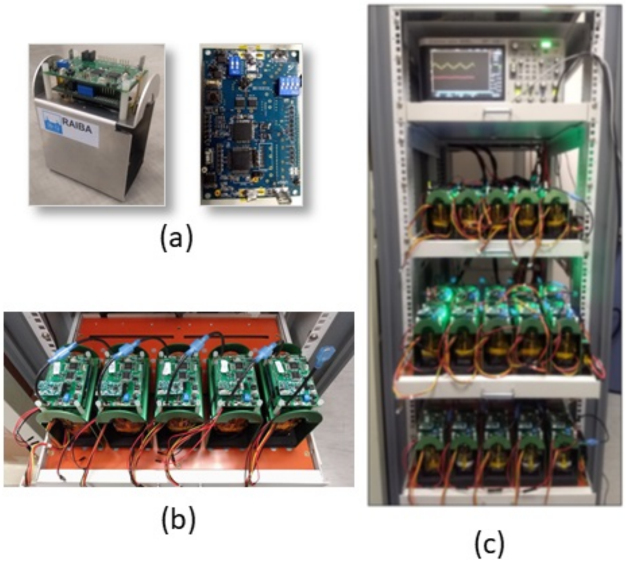


Figure 8: *The hardware implementation of the first* RAIBA *prototype, whose building block is a 4S2P battery module protected by an* ensemble/bypass *switch, shown in (a). Five such battery modules are connected in series to form a column, shown in (b), and the entire prototype contains five such columns connected in parallel and other measurement/control/protection circuits, and is housed in a rack, as shown in (c).*

A nice benefit of the *trace-based* approach to battery modeling is the model derived from measurements taken on a battery is tailored to and ages with the battery, and is thus more likely to better approximate the battery's ground truth than a pre-calibrated model.

5 Prototype Implementation

The first *RAIBA* prototype is a 5x5 array of battery modules, each of which in turn consists of 2 parallel-connected columns with each column comprising 4 Panasonic INR18650GA 3.4Ah batteries connected in series. Therefore, the entire prototype is composed of two hundred 18650 batteries and contains an energy capacity of 2.5KWh. As shown in Figure 8(a), associated with each battery module is an *enable/bypass* switch board and a cooling subsystem that provides thermal load management for the 4S2P (2 parallel-connected columns each having 4 series-connected batteries) batteries in the module. Five of these battery modules are connected in series to form a column, as shown in Figure 8(b), and five of these columns are connected in parallel and placed in a rack to form the complete battery array, as shown in Figure 8(c).

In addition to the battery array, the *RAIBA* prototype also contains a *RAIBA controller*, which is a Linux-based

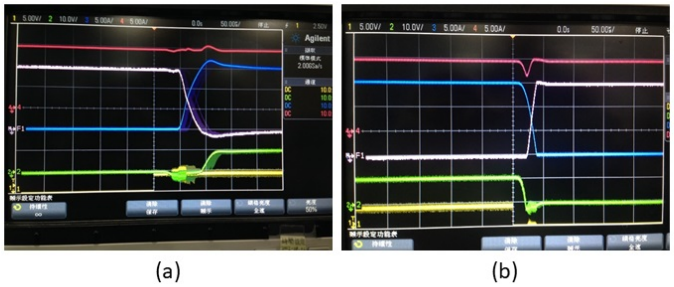
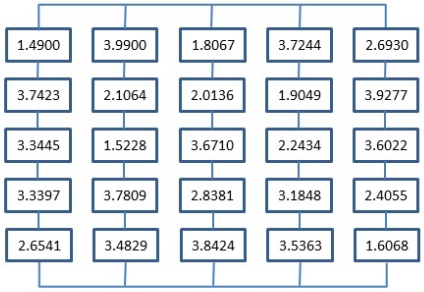


Figure 9: *The electric circuit behavior of the enable/bypass switch IC used in the current* RAIBA *prototype for (a) the transition from the bypass to enable mode, and (b) the transition from the enable to bypass mode. Red represents the current traversing the entire enable/bypass switch, White and Blue represent the currents traversing S1 and S2, respectively, and Green represent the voltage across the enable/bypass switch.*



1.4900	3.9900	1.8067	3.7244	2.6930
3.7423	2.1064	2.0136	1.9049	3.9277
3.3445	1.5228	3.6710	2.2434	3.6022
3.3397	3.7809	2.8381	3.1848	2.4055
2.6541	3.4829	3.8424	3.5363	1.6068

Figure 10: *The initial capacity of each battery in the array used in the test against five electric load traces*

Raspberry Pi 3 board that runs the Configuration Control algorithm based on electric load requests and battery status measurements, a *battery status measurement and communication module*, which uses a Linear Technology LTC6804-1 battery monitor to constantly measure the voltage, current, and temperature of each battery in the array, and an ATmega328P MCU to report these measurements in real time through the UART protocol to the RAIBA controller, and a *surge protector* that offers a safeguard mechanism to limit unexpected transient surge currents during battery array reconfiguration. The charger circuit and the DC/AC inverter for discharging are connected to the *surge protector* to charge and discharge the *RAIBA* prototype, respectively.

6 Performance Evaluation

6.1 Efficiency of Enable/Bypass Switch

Unlike digital logic switch or computer network switch, signals flowing on *RAIBA*'s enable/bypass switches represent energies to be delivered to and consumed by electric loads. Because these signals' magnitude is much larger and their transmission behavior is driven by instantaneous voltage level differences and thus largely analog, whether it is feasible to successfully implement an electrically safe enable/bypass switch that keeps the traversed electric current constant during the transitions between the enable mode and the bypass mode, raised serious doubts in the beginning of the *RAIBA* project. We have two IC implementations for the enable/bypass switch IC. The electric circuit behaviors of the version used in the current *RAIBA* prototype during the transitions of the enable and the bypass mode are shown in Figure 9. Red represents the current traversing the entire enable/bypass switch, White and Blue represent the currents traversing the two constituent switches S1 and S2 (shown in Figure 4(a)), respectively, and Green represent the voltage across the enable/bypass switch.

When the voltage drop across an enable/bypass switch is 16V and the running current is 15A, the total switch time is 50 μsec for this switch to go from the bypass to enable mode, and is 30 μsec for an enable/bypass switch to go from the enable to the bypass mode. The resulting power loss (P_{loss}) is 3mJ for the transition from the bypass to enable mode, and is 1.8mJ for the transition from the enable to bypass mode. Although there are still some glitches in the traversed current during the transitions, as indicated by the dips and bumps in the red curves in Figure 9 (a) and (b), these glitches are relatively small in magnitude and thus seamless to the electric loads.

The above results conclusively demonstrates that an implementation of an electrically safe enable/bypass switch IC which keeps the current traversing it constant during transitions not only is feasible, but also could be made highly efficient in terms of switch time and switching-induced energy loss.

6.2 Gains from Dynamic Reconfigurability

Intuitively, the more heterogeneous the batteries in a *RAIBA* array and the more fluctuated the electric load facing a *RAIBA* array, the higher performance gain *RAIBA*'s dynamic reconfigurability is expected to provide. To empirically assess the gain from *RAIBA*'s dynamic reconfigurability, we tested the first *RAIBA* prototype under a set of electric load traces until it cannot be discharged any further, and measure the total discharge service time, the percentage of wasted battery capacity at the end of the discharge cycle, and the number of enable/bypass mode transitions.

To reduce the amount of time required for each experiment run, we limited the capacity of each battery module to under 4Ah, but still used the entire 5x5 array. Because *RAIBA* is designed to minimize unnecessary energy waste when the constituent batteries in an array ex-

Electric Load	Discharge Service Time (hr)	Wasted Capacity (Ah)	Standard Deviation (Ah)	Transition Count
Simple	4.59 / 8.42 (83.4%)	35.5698 / 2.3747 (-93.3%)	0.8372 / 0.0428 (-94.9%)	0 / 3059
eBus	3.77 / 6.89 (82.9%)	35.5325 / 2.4114 (-93.2%)	0.8372 / 0.0695 (-91.7%)	0 / 1385
Data center	4.16 / 7.66 (84.0%)	35.5472 / 2.0324 (-94.3%)	0.8372 / 0.0931 (-88.9%)	0 / 2083
NYC	2.98 / 5.77 (93.4%)	35.4782 / 2.1170 (-94.3%)	0.8372 / 0.1040 (-87.6%)	0 / 2493
TaiPower	2.93 / 6.09 (108.1%)	35.4609 / 2.4612 (-93.1%)	0.8372 / 0.2510 (-70.0%)	0 / 545

Table 1: *Comparison between the* Fixed *configuration (left) and the* Reconfigurable *configuration (right) in terms of the total discharge service time, the total wasted capacity at the end, the standard deviation in battery capacity at the end, and the number of mode transitions, under five electric load traces. Numbers inside the parenthesis represent the percentage difference between the* Reconfigurable *configuration and the* Fixed *configuration.*

hibit diverse discharging characteristics or have different initial capacities, we focused below on a test case in which the initial capacities of the battery modules in the array vary from 1.49Ah to 3.99Ahm, as shown in Figure 10, where the standard deviation of the individual battery capacity across the array is 0.8377Ah. The total battery capacity of the entire array is 72.4544Ah, and the maximum power is 1790.3365W. To evaluate how the *RAIBA* prototype performs under different electric load patterns, we used the following five electric load traces that represent a wide variety of use cases:

- *Simple*: A constant load at the level of 120W.
- *eBus*: A simplified version of an electric load trace captured from an electric bus that consists of periods each of which includes a 5-sec accelerate phase of 200W, a 50-sec cruise phase of 150W, and a 10-sec decelerate phase of 100W.
- *Data center*: An electric load trace that was collected on 2016/08/02 from a 700+-server cloud computing data center within Industrial Technology Research Institute, and then scaled down in the power magnitude by a factor of 0.02.
- *NYC*: A scaled down version of the electric load trace for New York City on 2016/04/01 by a factor of $3 * 10^{-8}$.
- *Taipower*: A scaled down version of the electric load trace for the Northern part of Taiwan on 2016/03/29 by a factor of 10^{-7}.

We exercised each of the five electric load traces against the *RAIBA* prototype twice, once when we enabled the array's reconfigurability capability (*Reconfigurable* configuration), and the other time when we enabled it completely (*Fixed* configuration). At the beginning of each run, we charged each battery in the array according to the specification in Figure 10, and then discharged the array using a programmable electric load generator that drew power over time according to a given electric load trace. Each experiment run terminates when the *RAIBA* prototype can no longer continue servicing the corresponding electric load trace.

Table 1 shows the detailed comparisons between the *Fixed* and *Reconfigurable* configuration under the five electric load traces. For each and every of the five electric load traces, the *Reconfigurable* configuration outperforms the *Fixed* configuration in terms of the total discharge service time by between 82.9% and 108.1%. That is, for the same initial array condition and electric load trace, the *Reconfigurable* configuration lasts almost twice as long as the *Fixed* configuration. This gain mainly comes from the reduction in the imbalance of the batteries' residual capacity, as shown in the Standard Deviation column, which that indicates the *Reconfigurable* configuration reduces the standard deviation in residual battery capacity at the end of an experiment run by between 70% to 94.9% when compared with that of the *Fixed* configuration, which is roughly the same as the initial standard deviation because each battery contributes equally during the discharging process. Figure 11 shows visually how an array's batteries' residual capacities evolve over time under an *eBus* trace when dynamic reconfigurability is turned on and off. As expected, the residual capacities of an array's batteries converge over time towards a common value when *RAIBA*'s dynamic reconfigurability is enabled, but progress largely independently of one another when dynamic reconfigurability is disabled.

Hardware-based inter-battery capacity balancing, which transfers charge from larger-capacity batteries to lower-capacity batteries, and inevitably incurs energy loss. In contrast, *RAIBA* balances the residual capacities of an array's batteries by drafting different subsets of batteries to work at different time, and thus is more general because it could balance the residual capacities of those batteries that are not electrically connected, and more energy-efficient because it does not involve any movement of electric charges between batteries.

When the residual capacities of an array's batteries are more balanced, it is less likely that the array is forced to

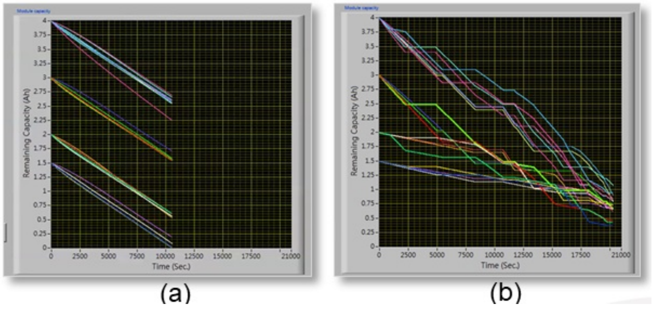


Figure 11: *The residual capacities of a* RAIBA *array's batteries evolve over time under an* eBus *trace when dynamic reconfigurability is turned (a) off and (b) on.*

terminate earlier on because of the capacity exhaustion of some batteries, and the amount of residual capacity lying unused and wasted at the end of each experiment run is expected to be smaller. The Wasted Capacity column of Table 1 shows that the wasted capacity at the end of an experiment run for the *Reconfigurable* configuration is less than 10% of that of the *Fixed* configuration for each of the five electric load traces. This demonstrates the *Reconfigurable* configuration's capability to eliminate energy waste due to fragmentation via more effective inter-battery capacity balancing.

That there is not much correlation between the Transition Count column and the Discharge Service Time column of Table 1 suggests that a higher number of mode transitions does not necessarily result in a higher gain in the total discharge service time. The gain also has a lot to do with the degree of mismatch between the demand patterns of an electric load trace and the energy profile that the *Fixed* configuration could offer.

The price of dynamic reconfigurability is the additional energy consumption due to transitions between the enable and the bypass mode. However, the associated energy consumption is rather miniscule. For example, even if a discharge cycle requires 3000 mode transitions, as in the case of the *Simple* trace in Table 1, the total energy consumption is about $3000 * 3mJ = 9J$, which is about 0.001% of the total energy capacity of the 5x5 battery array used in the test.

7 Related Work

Song Ci et al. [4] provides a detailed survey of the reconfigurable battery techniques, including various reconfigurable battery array designs proposed in the literature, their management and fault tolerance properties, and the design considerations of the associated battery management mechanisms. Baronti et al. [3] and Miyatake et al. [20] explored the effective capacity of a battery array with different inter-battery connectivity configurations. Baronti et al. [2] explored the design space for the bypass/enable switch module. Kim and Shin [15] first proposed a dynamically reconfigurable architecture for large-scale battery arrays used in electric vehicles in order to tolerate battery cell failures. Jin and Shin [13] followed up on [15] with the development of battery pack sizing and reconfiguration algorithms. Kim et al. [16] built the first small-scale (6x3) reconfigurable battery array called self-X, which aimed to tolerate battery failures, balance the capacities among batteries and optimize energy conversion efficiency. He et al. [9] took into account battery conditions, particularly state of health, to dynamically reconfigure a battery array to maximize its total capacity. He et al. [10] improved the performance of charging operations by leveraging various battery state information to best exploit dynamic reconfigurability. He et al. [11] used dynamic reconfigurability to allow weaker cells to rest longer so as to balance inter-battery capacity and increase the battery array's effective capacity.

Badam et al. [1] proposed a software-defined battery system that includes batteries of different charging/discharging characteristics and provides an API for the control software to use the most appropriate batteries to service given electric loads. While dynamic reconfigurability was originally proposed for large-scale battery arrays, Visairo and Kumar [21] and Ci et al. [5] explored the effectiveness of applying dynamic reconfigurability to portable and mobile devices.

RAIBA differs from the research efforts described in the following ways. First, RAIBA features a real implementation of a switching IC that is able to enable and bypass a battery of a battery array while keeping the traversing current constant and the array continuing operating. This real-time dynamic reconfigurability makes it possible to apply RAIBA to applications beyond stand-by energy storage systems, such as electric vehicles. Second, RAIBA supports a dynamic battery array reconfiguration algorithm that takes into account the capacity/state of each battery and the target electric load, and produces in real time a battery array configuration that meets the energy needs of the target load while minimizing unnecessary energy waste due to inter-battery balancing and power conversion. Third, RAIBA adopts a trace-based battery model that removes the need for batteries with homogeneous quality, and is able to accommodate batteries that age over times.

8 Conclusion

This work proposes a *RAIBA* approach to using retired batteries to build cost-effective energy storage systems that make up for the intermittent nature of renewable energy generation systems. The key idea in *RAIBA* is to use dynamic reconfigurability to make the best of heterogeneity in retired batteries, so as to enable continued operations even in the presence of individual battery failures, maximize the energy output of each charge/discharge cycle, and minimize energy loss due to conversion, analog inter-battery capacity balancing and resource fragmentation. The paper describes the design, implementation and evaluation of a fully operational *RAIBA* -1 prototype. More specifically, this paper makes the following contributions to the energy storage management area:

- A taxonomic framework for analyzing varying degrees of flexibility of dynamically reconfiguring the inter-battery connectivity of large-scale battery arrays at run time,
- The first successful and efficient implementation of an *enable/bypass* switch IC that keeps the traversed current constant during transitions between the bypass and enable mode, and
- The completion of a fully operational software-defined virtualized battery array prototype, and an empirical demonstration of the efficacy of its dynamic reconfigurability in increasing the effective discharge service time by more than 80% for a variety of electric load traces.

Notes

[1] Authors are listed in the alphabetical order of their last names.

References

[1] Anirudh Badam, Ranveer Chandra, Jon Dutra, Anthony Ferrese, Steve Hodges, Pan Hu, Julia Meinershagen, Thomas Moscibroda, Bodhi Priyantha, and Evangelia Skiani. Software defined batteries. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, pages 215–229, October 2015.

[2] Federico Baronti, Gabriele Fantechi, Roberto Roncella, and Roberto Saletti. Design of a module switch for battery pack reconfiguration in high-power applications. In *Proceedings of IEEE International Symposium on Industrial Electronics*, pages 1330–1335, May 2012.

[3] Federico Baronti, Roberto Di Rienzo, Nicola Papazafiropulos, Roberto Roncella, and Roberto Saletti. Investigation of series-parallel connections of multi-module batteries for electrified vehicles. In *Proceedings of IEEE International Electric Vehicle Conference*, Dec. 2014.

[4] Song Ci, Ni Lin, and Dalei Wu. Reconfigurable battery techniques and systems: A survey. *IEEE Acess*, 4:1175–1189, 2016.

[5] Song Ci, Jiucai Zhang, Hamid Sharif, and Mahmoud Alahmad. Dynamic reconfigurable multi-cell battery: A novel approach to improve battery performance. In *Proceedings of Applied Power Electronics Conference and Exposition (APEC)*, pages 439–442, 2012.

[6] Tzi-cker Chiueh, Chia-Ming Chang, Welkin Ling, and Shih-Hao Liang. Power management method for low capacity state of electro-chemical batteries. In *US Patent No. US9229510*, January 2016.

[7] Tzi-cker Chiueh, Shih-Hao Liang, Kai-Cheung Juang, and Shou-Hung Ling. Battery system and control method thereof. In *US Publication No. US20170133865 A1*, May 2017.

[8] Stephen D. Comello, Gunther Glenk, and Stefan Reichelstein. Levelized cost of electricity calculator: A user guide. *Stanford School of Business, Sustaintable Energy Initiative*, May 2017. `http://stanford.edu/dept/gsb_circle/cgibin/sustainableEnergy/GSB_LCOE_User%20Guide_0517.pdf`.

[9] Liang He, Yu Gu, Ting Zhu, Cong Liu, and Kang G. Shin. Share: Soh-aware reconfiguration to enhance deliverable capacity of large-scale battery packs. In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems (ICCPS '15)*, pages 169–178, April 2015.

[10] Liang He, Eugene Kim, and Kang G. Shin. -aware charging of lithium-ion battery cells. In *Proceedings of 7th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS 2016)*, pages 26:1–26:10, April 2016.

[11] Liang He, Eugene Kim, and Kang G. Shin. Resting weak cells to improve battery pack's capacity delivery via reconfiguration. In *Proceedings of the Seventh International Conference on Future Energy Systems*, pages 8:1–8:11, June 2016.

[12] Chien-Tung Hsu, Shou-Hung Ling, Kai-Cheung Juang, Tzi-cker Chiueh, and Chuan-Yu Cho. Programmable battery source architecture and method thereof. In *US Publication No. US 20160164315 A1*, June 2016.

[13] Fangjian Jin and Kang G. Shin. Pack sizing and reconfiguration for management of large-scale batteries. In *Proceedings of IEEE/ACM Third International Conference on Cyber-Physical Systems (ICCPS 2012)*, pages 138–147, 2012.

[14] Yahiko Kambayashi. Logic design of programmable logic arrays. *IEEE Transactions on Computers*, 28:586–590, 1979.

[15] Hahnsang Kim and Kang G. Shin. On dynamic reconfiguration of a large-scale battery system. In *Proceedings of 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2009)*, pages 87–96, April 2009.

[16] Taesic Kim, Wei Qiao, and Liyan Qu. Power electronics-enabled self-x multicell batteries: a design toward smart batteries. *IEEE Transactions on Power Electronics*, 27:4273–4283, 2012.

[17] Shih-Hao Liang, Tzi-cker Chiueh, and Welkin Ling. Go gentle into the good night via controlled battery discharging. In *Proceedings of the 6th Asia-Pacific Workshop on Systems, (APSys 2015)*, July 2015.

[18] David Linden and Thomas B. Reddy. *Hand book of batteries (3rd edition)*. McGraw-Hill, 2002.

[19] Welkin Ling, Chein-Chung Sun, and Chiou-Chu Lai. Method for checking and modulating battery capacity and power based on discharging/charging characteristics. In *US Patent No. US8823325*, September 2014.

[20] So Miyatake, Yoshihiko Susuki, Takashi Hikihara, Syuichi Itoh, and Kenichi Tanaka. Discharge characteristics of multicell lithium-ion battery with nonuniform cells. *Journal of Power Sources*, 241:736–743, 2015.

[21] H. Visairo and P. Kumar. A reconfigurable battery pack for improving power conversion efficiency in portable devices. In *Proceedings of the 7th International Caribbean Conference on Devices, Circuits and Systems*, April 2008.

CNTR: Lightweight OS Containers

Jörg Thalheim, Pramod Bhatotia
University of Edinburgh

Pedro Fonseca
University of Washington

Baris Kasikci
University of Michigan

Abstract

Container-based virtualization has become the de-facto standard for deploying applications in data centers. However, deployed containers frequently include a wide-range of tools (e.g., debuggers) that are not required for applications in the common use-case, but they are included for rare occasions such as in-production debugging. As a consequence, containers are significantly larger than necessary for the common case, thus increasing the build and deployment time.

CNTR[1] provides the *performance* benefits of lightweight containers and the *functionality* of large containers by splitting the traditional container image into two parts: the "fat" image — containing the tools, and the "slim" image — containing the main application. At run-time, CNTR allows the user to efficiently deploy the "slim" image and then expand it with additional tools, when and if necessary, by dynamically attaching the "fat" image.

To achieve this, CNTR transparently combines the two container images using a new nested namespace, without any modification to the application, the container manager, or the operating system. We have implemented CNTR in Rust, using FUSE, and incorporated a range of optimizations. CNTR supports the full Linux filesystem API, and it is compatible with all container implementations (i.e., Docker, rkt, LXC, systemd-nspawn). Through extensive evaluation, we show that CNTR incurs reasonable performance overhead while reducing, on average, by 66.6% the image size of the Top-50 images available on Docker Hub.

1 Introduction

Containers offer an appealing, lightweight alternative to VM-based virtualization (e.g., KVM, VMware, Xen) that relies on process-based virtualization. Linux, for instance, provides the `cgroups` and `namespaces` mechanisms that enable strong performance and security isolation between containers [24]. Lightweight virtualization is fundamental to achieve high efficiency in virtualized datacenters and enables important use-cases, namely just-in-time deployment of applications. Moreover, containers significantly reduce operational costs through higher consolidation density and power minimization, especially in multi-tenant environments. Because of all these advantages, it is no surprise that containers have seen wide-spread adoption by industry, in many cases replacing altogether traditional virtualization solutions [17].

Despite being lightweight, deployed containers often include a wide-range of tools such as shells, editors, coreutils, and package managers. These additional tools are usually not required for the application's core function — the common operational use-case — but they are included for management, manual inspection, profiling, and debugging purposes [64]. In practice, this significantly *increases container size* and, in turn, translates into slower container deployment and inefficient datacenter resource usage (network bandwidth, CPU, RAM and disk). Furthermore, larger images degrade container deployment time [52, 44]. For instance, previous work reported that downloading container images account for 92% of the deployment time [52]. Moreover, a larger code base directly affects the reliability of applications in datacenters [50].

Given the impact of using large containers, users are discouraged from including additional tools that would otherwise simplify the process of debugging, deploying, and managing containers. To mitigate this problem, Docker has recently adopted smaller run-times but, unfortunately, these efforts come at the expense of compatibility problems and have limited benefits [13].

To quantify the practical impact of additional tools on the container image size, we employed Docker Slim [11] on the 50 most popular container images available on the Docker Hub repository [10]. Docker Slim uses a combination of static and dynamic analyses to generate smaller-sized container images, in which, only files needed by the core application are included in the final image. The results of this experiment (see Figure 5) are

[1] Read it as "center".

encouraging: we observed that by excluding unnecessary files from typical containers it is possible to reduce the container size, on average, by 66.6%. Similarly, others have found that a only small subset (6.4%) of the container images is read in the common case [53].

CNTR addresses this problem[2] by building lightweight containers that still remain fully functional, even in uncommon use-cases (e.g., debugging and profiling). CNTR enables users to deploy the application and its dependencies, while the additional tools required for other use-cases are supported by expanding the container "on-demand", during runtime (Figure 1 (a)). More specifically, CNTR splits the traditional container image into two parts: the "fat" image containing the rarely used tools and the "slim" image containing the core application and its dependencies.

During runtime, CNTR allows the user of a container to efficiently deploy the "slim" image and then expand it with additional tools, when and if necessary, by dynamically attaching the "fat" image. As an alternative to using a "fat" image, CNTR allows tools from the *host* to run inside the container. The design of CNTR simultaneously preserves the performance benefits of lightweight containers and provides support for additional functionality required by different application workflows.

The key idea behind our approach is to create a new *nested namespace* inside the application container (i.e., "slim container"), which provides access to the resources in the "fat" container, or the host, through a FUSE filesystem interface. CNTR uses the FUSE system to combine the filesystems of two images without any modification to the application, the container implementation, or the operating system. CNTR selectively redirects the filesystem requests between the mount namespace of the container (i.e., what applications within the container observe and access) and the "fat" container image or the host, based on the filesystem request path. Importantly, CNTR supports the full Linux filesystem API and all container implementations (i.e., Docker, rkt, LXC, systemd-nspawn).

We evaluated CNTR across three key dimensions: (1) *functional completeness* – CNTR passes 90 out of 94 (95.74%) `xfstests` filesystem regression tests [14] supporting applications such as SQLite, Postgres, and Apache; (2) *performance* – CNTR incurs reasonable overheads for the `Phoronix` filesystem benchmark suite [18], and the proposed optimizations significantly improve the overall performance; and lastly, (3) *effectiveness* – CNTR's approach on average results in a 66.6% reduction of image size for the Top-50 images available on Docker hub [10]. We have made publicly available the CNTR implementation along with the experimental setup [6].

2 Background and Motivation

2.1 Container-Based Virtualization

Containers consist of a lightweight, process-level form of virtualization that is widely used and has become a cornerstone technology for datacenters and cloud computing providers. In fact, all major cloud computing providers (e.g., Amazon [2], Google [16] and Microsoft [4]) offer Containers as a Service (CaaS).

Container-based virtualization often relies on three key components: (1) the OS mechanism that enforces the process-level isolation (e.g., the Linux `cgroups` [41] and `namespaces` [40] mechanisms), (2) the application packaging system and runtime (e.g., Docker [9], Rkt [38]), and (3) the orchestration manager that deploys, distributes and manages containers across machines (e.g., Docker Swarm [12], Kubernetes [22]). Together, these components enable users to quickly deploy services across machines, with strong performance and security isolation guarantees, and with low-overheads.

Unlike VM-based virtualization, containers do not include a guest kernel and thus have often smaller memory footprint than traditional VMs. Containers have important advantages over VMs for both users and data centers:

1. **Faster deployment.** Containers are transferred and deployed faster from the registry [44].
2. **Lower resource usage.** Containers consume fewer resources and incur less performance overhead [62].
3. **Lower build times.** Containers with fewer binaries and data can be rebuilt faster [64].

Unfortunately, containers in practice are still unnecessarily large because users are forced to decide which auxiliary tools (e.g. debugging, profiling, etc.) should be included in containers at *packaging-time*. In essence, users are currently forced to strike a balance between lightweight containers and functional containers, and end up with containers that are neither as light nor as functional as desirable.

2.2 Traditional Approaches to Minimize Containers

The container-size problem has been a significant source of concern to users and developers. Unfortunately, existing solutions are neither practical nor efficient.

An approach that has gained traction, and has been adopted by Docker, consists of packing containers using smaller base distributions when building the container runtime. For instance, most of Docker's containers are now based on the Alpine Linux distribution [13], resulting in smaller containers than traditional distributions. Alpine Linux uses the `musl` library, instead of `libc`, and bundles `busybox` , instead of `coreutils` — these differences enable a smaller container runtime but at the expense of compatibility problems caused by runtime differences. Further, the set of tools included is still restricted and fundamentally does not help users when less common auxiliary tools are required (e.g., custom debugging tools).

[2] Note that Docker Slim [11] does not solve the problem; it simply identifies the files not required by the application, and excludes them from the container, but it does not allow users to access those files at run-time.

The second approach to reduce the size of containers relies on union filesystems (e.g., UnionFS [60]). Docker, for instance, enables users to create their containers on top of commonly-used base images. Because such base images are expected to be shared across different containers (and already deployed in the machines), deploying the container only requires sending the diff between the base image and the final image. However, in practice, users still end up with multiple base images due to the use of different base image distributions across different containers.

Another approach that has been proposed relies on the use of *unikernels* [57, 58], a single-address-space image constructed from a *library OS* [61, 49, 65]. By removing layers of abstraction (e.g., processes) from the OS, the unikernel approach can be leveraged to build very small virtual machines—this technique has been considered as containerization because of its low overhead, even though it relies on VM-based virtualization. However, unikernels require additional auxiliary tools to be statically linked into the application image; thus, it leads to the same problem.

2.3 Background: Container Internals

The container abstraction is implemented by a userspace container run-time, such as Docker [9], rkt [38] or LXC [37]. The kernel is only required to implement a set of per-process isolation mechanisms, which are inherited by child processes. This mechanism is in turn leveraged by container run-times to implement the actual container abstraction. For instance, applications in different containers are isolated and have all their resources bundled through their own filesystem tree. Crucially, the kernel allows the partitioning of system resources, for a given process, with very low performance overhead thus enabling efficient process-based virtualization.

The Linux operating system achieves isolation through an abstraction called namespaces. Namespaces are modular and are applied to individual processes inherited by child processes. There are seven namespaces to limit the scope what a process can access (e.g., filesystem mountpoints, network interfaces, or process IDs[40]).

During the container startup, by default, namespaces of the host are unshared. Hence, processes inside the container only see files from their filesystem image (see Figure 1 (a)) or additional volumes, that have been statically added during setup. New mounts on the host are not propagated to the container since by default, the container runtime will mount all mount points as private.

2.4 Use-cases of CNTR

We envision three major use cases for CNTR that cover three different debugging/management scenarios:

Container to container debugging in production. CNTR enables the isolation of debugging and administration tools in *debugging containers* and allows application containers to use debugging containers on-demand. Consequently, application containers become leaner, and the isolation of debugging/administration tools from applications allows users to have a more consistent debugging experience. Rather than relying on disparate tools in different containers, CNTR allows using a single debugging container to serve many application containers.

Host to container debugging. CNTR allows developers to use the debugging environments (e.g., IDEs) in their host machines to debug containers that do not have these environments installed. These IDEs can sometimes take several gigabytes of disk space and might be not even compatible with the distribution of the container image is based on. Another benefit of using CNTR in this context is that development environments and settings can be also efficiently shared across different containers.

Container to host administration and debugging. Container-oriented Linux distributions such as CoreOS [8] or RancherOS [30] do not provide a package manager and users need to extend these systems by installing containers even for basic system services. CNTR allows a user of a privileged container to access the root filesystem of the host operating system. Consequently, administrators can keep tools installed in a debug container while keeping the host operating system's filesystem lean.

3 Design

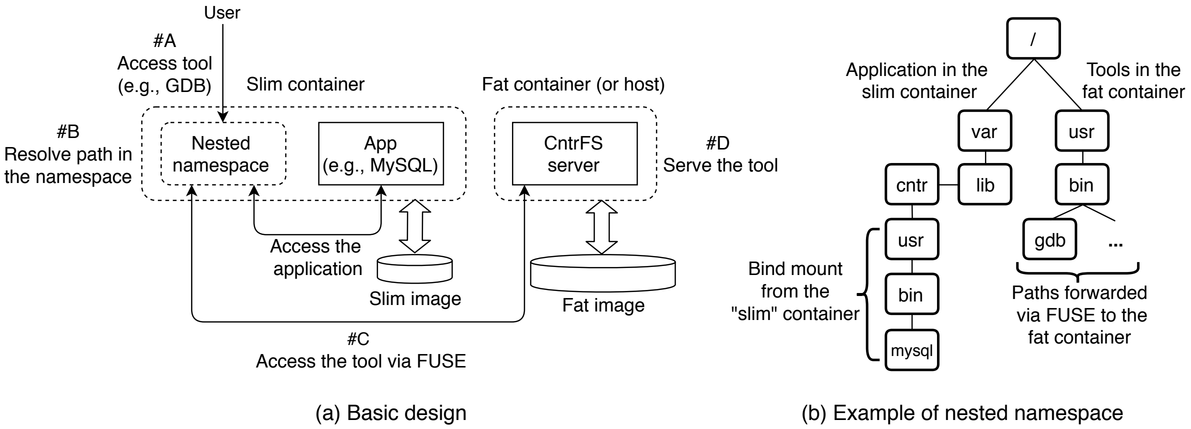
In this section, we present the detailed design of CNTR.

3.1 System Overview

Design goals. CNTR has the following design goals:

- *Generality:* CNTR should support a wide-range of workflows for seamless management and problem diagnosis (e.g., debugging, tracing, profiling).
- *Transparency:* CNTR should support these workflows without modifying the application, the container manager, or the operating system. Further, we want to be compatible with all container implementations.
- *Efficiency:* Lastly, CNTR should incur low performance overheads with the split-container approach.

Basic design. CNTR is composed of two main components (see Figure 1 (a)): a nested namespace, and the CNTRFS filesystem. In particular, CNTR combines slim and fat containers by creating a new *nested namespace* to merge the namespaces of two containers (see Figure 1 (b)). The nested namespace allows CNTR to selectively break the isolation between the two containers by transparently redirecting the requests based on the accessed path. CNTR achieves this redirection using the CNTRFS filesystem. CNTRFS is mounted as the root filesystem (`/`), and the application filesystem is remounted to another path (`/var/lib/cntr`) in the nested namespace. CNTRFS implements a filesystem in userspace (FUSE), where the CNTRFS server handles the requests for auxiliary tools installed on the fat container (or on the host).



(a) Basic design

(b) Example of nested namespace

Figure 1: Overview of CNTR

At a high-level, CNTR connects with the CNTRFS server via the generic FUSE kernel driver. The kernel driver simply acts as a proxy between processes accessing CNTRFS, through Linux VFS, and the CNTRFS server running in userspace. The CNTRFS server can be in a different mount namespace than the nested namespace, therefore, CNTR establishes a proxy between two mount namespaces through a request/response protocol. This allows a process that has all its files stored in the fat container (or the host) to run within the mount namespace of the slim container.

Cntr workflow. CNTR is easy to use. The user simply needs to specify the name of the "slim" container and, in case the tools are in another container, the name of the "fat" container. CNTR exposes a shell to the user that has access to the resources of the application container as well as the resources forwarded from the fat container.

Figure 1 (a) explains the workflow of CNTR when a user requests to access a tool from the slim container (#A): CNTR transparently resolves the requested path for the tool in the nested namespace (#B). Figure 1 (b) shows an example of CNTR's nested namespace, where the requested tool (e.g., gdb) is residing in the fat container. After resolving the path, CNTR redirects the request via FUSE to the fat container (#C). Lastly, CNTRFS serves the requested tool via the FUSE interface (#D). Behind the scenes, CNTR executes the following steps:

1. *Resolve container name to process ID and get container context.* CNTR resolves the name of the underlying container process IDs and then queries the kernel to get the complete execution context of the container (container namespaces, environment variables, capabilities, ...).
2. *Launch the* CNTRFS *server.* CNTR launches the CNTRFS server. CNTR launches the server either directly on the host or inside the specified "fat" container containing the tools image, depending on the settings that the user specified.
3. *Initialize the tools namespace.* Subsequently, CNTR attaches itself to the application container by setting up a nested mount namespace within the namespace of the application container. CNTR then assigns a forked process to the new namespace. Inside the new namespace, the CNTR process proceeds to mount CNTRFS, providing access to files that are normally out of the scope of the application container.
4. *Initiate an interactive shell.* Based on the configuration files within the debug container or on the host, CNTR executes an interactive shell, within the nested namespace, that the user can interact with. CNTR forwards its input/output to the user terminal (on the host). From the shell, or through the tools it launches, the user can then access the application filesystem under `/var/lib/cntr` and the tools filesystem in `/`. Importantly, tools have the same view on system resources as the application (e.g., `/proc`, `ptrace`). Furthermore, to enable the use of graphical applications, CNTR forwards Unix sockets from the host/debug container.

3.2 Design Details

This section explains the design details of CNTR.

3.2.1 Step #1: Resolve Container Name and Obtain Container Context

Because the kernel has no concept of a container name or ID, CNTR starts by resolving the container name, as defined by the used container manager, to the process IDs running inside the container. CNTR leverages wrappers based on the container management command line tools to achieve this translation and currently, it supports Docker, LXC, rkt, and systemd-nspawn.

After identifying the process IDs of the container, CNTR gathers OS-level information about the container namespace. CNTR reads this information by inspecting the `/proc` filesystem of the main process within the container.

This information enables CNTR to create processes inside the container in a *transparent* and *portable* way.

In particular, CNTR gathers information about the container namespaces, cgroups (resource usage limits), mandatory access control (e.g., AppArmor [26] and SELinux [19] options), user ID mapping, group ID mapping, capabilities (fine-grained control over super-user permissions), and process environment options. Additionally, CNTR could also read the seccomp options, but this would require non-standard kernel compile-time options and generally has limited value because seccomp options have significant overlap with the capability options. CNTR reads the environment variables because they are heavily used in containers for configuration and service discovery [36].

Before attaching to the container, in addition, to gather the information about the container context, the CNTR process opens the FUSE control socket (`/dev/fuse`). This file descriptor is required to mount the CNTRFS filesystem, after attaching to the container.

3.2.2 Step #2: Launch the CNTRFS Server

The CNTRFS is executed either directly on the host or inside the "fat" container, depending on the option specified by the user (i.e., the location of the tools). In the host case the CNTRFS server simply runs like a normal host process.

In case the user wants to use tools from the "fat" container, the CNTRFS process forks and attaches itself to the "fat" container. Attaching to the "fat" container is implemented by calling the `setns()` system call, thereby assigning the child process to the container namespace that was collected in the previous step.

After initialization, the CNTRFS server waits for a signal from the nested namespace (Step #3) before it starts reading and serving the FUSE requests (reading before an unmounted FUSE filesystem would otherwise return an error). The FUSE requests then will be read from the `/dev/fuse` file descriptor and redirected to the filesystem of the server namespace (i.e., host or fat container).

3.2.3 Step #3: Initialize the Tools Namespace

CNTR initializes the tool namespace by first attaching to the container specified by the user—the CNTR process forks and the child process assigns itself to the cgroup, by appropriately setting the `/sys/` option, and namespace of the container, using the `setns()` system call.

After attaching itself to the container, CNTR creates a new nested namespace, and marks all mountpoints as private so that further mount events (regarding the nested namespace) are not propagated back to the container namespace. Subsequently, CNTR creates a new filesystem hierarchy for the nested namespace, mounting the CNTRFS in a temporary mountpoint (`TMP/`).

Within the nested namespace, the child process mounts CNTRFS, at `TMP/`, and signals the parent process (running outside of the container) to start serving requests. Signalling between the parent and child CNTR processes is implemented through a shared Unix socket.

Within the nested namespace, the child process remounts all pre-existing mountpoints, from the application container, by moving them from `/` to `TMP/var/lib/cntr`. Note that the application container is not affected by this since all mountpoints are marked as private.

In addition, CNTR also mounts special container-specific files from the application over files from the tools or host (using bind mount [42]). The special files include the pseudo filesystems `procfs` (`/proc`), ensuring the tools can access the container application, and devtmpfs (`/dev`), containing block and character devices that have been made visible to our container. Furthermore, we bind mount a set of configuration files from the application container into the temporary directory (e.g., `/etc/passwd`, and `/etc/hostname`).

Once the new filesystem hierarchy has been created in the temporary directory, CNTR atomically executes a `chroot` turning the temporary directory (`TMP/`) into the new root directory (`/`).

To conclude the container attachment and preserve the container isolation guarantees, CNTR updates the remaining properties of the nested namespace: (1) CNTR drops the capabilities by applying the AppArmor/SELinux profile and (2) CNTR applies all the environment variables that were read from the container process; with the exception of `PATH` – the `PATH` is instead inherited from the debug container since it is often required by the tools.

3.2.4 Step #4: Start Interactive Shell

Lastly, CNTR launches an interactive shell within the nested namespace, enabling users to execute the tools. CNTR forwards the shell I/O using a pseudo-TTY, and supports graphical interface using Unix sockets forwarding.

Shell I/O. Interactive shells perform I/O through standard file descriptors (i.e., stdin, stdout, and stderr file descriptors) that generally refer to terminal devices. For isolation and security reasons, CNTR prevents leaking the terminal file descriptors of the host to a container by leveraging pseudo-TTYs – the pseudo-TTY acts as a proxy between the interactive shell and the user terminal device.

Unix socket forwarding. CNTR forwards connections to Unix sockets, e.g., the X11 server socket and the D-Bus daemon running on the host. Unix sockets are also visible as files in our FUSE. However, since our FUSE has inode numbers that are different from the underlying filesystem, the kernel does not associate them with open sockets in the system. Therefore, we implemented a socket proxy that runs an efficient event loop based on epoll. It uses the splice syscall to move data between clients in the application container and servers listening on Unix sockets in the debug container/host.

3.3 Optimizations

We experienced performance slowdown in CNTRFS when we measured the performance using the Phoronix benchmark suite [18] (§5.2). Therefore, we incorporated the following performance optimizations in CNTR.

Caching: Read and writeback caches. The major performance improvement gain was by allowing the FUSE kernel module to cache data returned from the read requests as well as setting up a writeback buffer for the writes. CNTR avoids automatic cache invalidation when a file is opened by setting the `FOPEN_KEEP_CACHE` flag. Without this flag the cache cannot be effectively shared across different processes. To allow the FUSE kernel module to batch smaller write requests, we also enable the writeback cache by specifying the `FUSE_WRITEBACK_CACHE` flag at the mount setup time. This optimization sacrifices write consistency for performance by delaying the sync operation. However, we show that it still performs correctly according to the POSIX semantics in our regression experiments (see § 5.1).

Multithreading. Since the I/O operations can block, we optimized the CNTRFS implementation to use multiple threads. In particular, CNTR spawns independent threads to read from the CNTRFS file descriptor independently to avoid contentions while processing the I/O requests.

Batching. In addition to caching, we also batch operations to reduce the number of context switches. In particular, we apply the batching optimization in three places: (a) pending inode lookups, (b) forget requests, and (c) concurrent read requests.

Firstly, we allow concurrent inode lookups by applying `FUSE_PARALLEL_DIROPS` option on mount. Secondly, the operating system sends `forget` requests, when inodes can be freed up by CNTRFS. The kernel can batch a forget intent for multiple inodes into a single request. In CNTR we have also implemented this request type. Lastly, we set `FUSE_ASYNC_READ` to allow the kernel to batch multiple concurrent read requests at once to improve the responsiveness of read operations.

Splicing: Read and write. Previous work suggested the use of splice reads and writes to improve the performance of FUSE [66]. The idea behind splice operation is to avoid copying data from and to userspace. CNTR uses splice for read operations. Therefore, the FUSE userspace process moves data from the source file descriptor into a kernel pipe buffer and then to the destination file descriptor with the help of the `splice` syscall. Since splice does not actually copy the data but instead remaps references in the kernel, it reduces the overhead.

We also implemented a splice write optimization. In particular, we use a pipe as a temporary storage, where the data is part of the request, and the data is not read from a file descriptor. However, FUSE does not allow to read the request header into userspace without reading the attached data. Therefore, CNTR has to move the whole request to a kernel pipe first in order to be able to read the request header separately. After parsing the header it can move the remaining data to its designated file descriptor using the splice operation. However, this introduces an additional context switch, and slowdowns all FUSE operations since it is not possible to know in advance if the next request will be a write request. Therefore, we decided not to enable this optimization by default.

4 Implementation

To ensure portability and maintainability, we decided not to rely on container-specific APIs, since they change quite often. Instead, we built our system to be as generic as possible by leveraging more stable operating system interfaces. Our system implementation supports all major container types: Docker, LXC, systemd-nspawn and rkt. CNTR's implementation resolves container names to process ids. Process ids are handled in an implementation-specific way. On average, we changed only 70 LoCs for each container implementation to add such container-specific support for CNTR.

At a high-level, our system implementation consists of the following four components:

- *Container engine* (1549 LoC) analyzes the container that a user wants to attach to. The container engine also creates a nested mount namespace, where it starts the interactive shell.
- CNTRFS (1481 LoC) to serve the files from the fat container. We implemented CNTRFS based on Rust-FUSE [33]. We extended Rust-FUSE to be able to mount across mount namespaces and without a dedicated FUSE mount executable.
- A *pseudo TTY* (221 LoC) to connect the shell input/output with the user terminal.
- A *socket proxy* (400 LoC) to forward the Unix socket connection between the fat (or the host) and slim containers for supporting X11 applications.

All core system components of CNTR were implemented in Rust (total 3651 LoC). To simplify deployment, we do not depend on any non-Rust libraries. In this way, we can compile CNTR as a ∼1.2MB single self-contained static executable by linking against musl-libc [23]. This design is imperative to ensure that CNTR can run on container-optimized Linux distributions, such as CoreOS [8] or RancherOS [30], that do not have a package manager to install additional libraries.

Since CNTR makes heavy use of low-level filesystem system calls, we have also extended the Rust ecosystem with additional 46 system calls to support the complete Linux filesystem API. In particular, we extended the nix Rust library [34], a library wrapper around the Linux/`POSIX` API. The changes are available in our fork [29].

5 Evaluation

In this section, we present the experimental evaluation of CNTR. Our evaluation answers the following questions.

1. Is the implementation complete and correct? (§5.1)
2. What are the performance overheads and how effective are the proposed optimizations? (§5.2)
3. How effective is the approach to reducing container image sizes? (§5.3)

5.1 Completeness and Correctness

We first evaluate the completeness and correctness claim of the CNTR implementation. The primary goal is to evaluate whether CNTR implements the same features (completeness) as required by the underlying filesystem, and it follows the same POSIX semantics (correctness).

Benchmark: xfstests regression test suite. For this experiment, we used the xfstests [14] filesystem regression test suite. The xfstests suite was originally designed for the XFS filesystem, but it is now widely used for testing all of Linux's major filesystems. It is regularly used for quality assurance before applying changes to the filesystem code in the Linux kernel. xfstests contains tests suites to ensure correct behavior of all filesystem related system calls and their edge cases. It also includes crash scenarios and stress tests to verify if the filesystem correctly behaves under load. Further, it contains many tests for bugs reported in the past.

Methodology. We extended xfstests to support mounting CNTRFS. For running tests, we mounted CNTRFS on top of tmpfs, an in-memory filesystem. We run all tests in the generic group once.

Experimental results. xfstests consists of 94 unit tests that can be grouped into the following major categories: *auto*, *quick*, *aio*, *prealloc*, *ioctl*, and *dangerous*.

Overall, CNTR passed 90 out of 94 (95.74%) unit tests in xfstests. Four tests failed due minor implementation details that we currently do not support. Specifically, these four unit tests were automatically skipped by xfstests because they expected our filesystem to be backed by a block device or expected some missing features in the underlying tmpfs filesystem, e.g. copy-on-write ioctl. We next explain the reasons for the failed four test cases:

1. **Test #375** failed since SETGID bits were not cleared in chmod when the owner is not in the owning group of the access control list. This would require manual parsing and interpreting ACLs in CNTR. In our implementation, we delegate POSIX ACLs to the underlying filesystem by using setfsuid/setfsguid on inode creation.
2. **Test #228** failed since we do not enforce the per-process file size limits (RLIMIT_FSIZE). As replay file operations and RLIMIT_FSIZE of the caller is not set or enforced in CNTRFS, this has no effect.
3. **Test #391** failed since we currently do not support the direct I/O flag in open calls. The support for direct I/O and mmap in FUSE is mutually exclusive. We chose mmap here, since we need it to execute processes. In practice, this is not a problem because not all docker drivers support this feature, including the popular filesystems such as overlayfs and zfs.
4. **Test #426** failed since our inodes are not exportable. In Linux, a process can get inode references from filesystems by the name_to_handle_at system call. However, our inodes are not persisted and are dynamically requested and destroyed by the operating system. If the operating system no longer uses them, they become invalid. Many container implementations block this system call as it has security implications.

To summarize, the aforementioned failed test cases are specific to our current state of the implementation, and they should not affect most real-world applications. As such, these features are not required according to the POSIX standard, but, they are Linux-specific implementation details.

5.2 Performance Overheads and Optimizations

We next report the performance overheads for CNTR's split-containers approach (§5.2.1), detailed experimental results (§5.2.2), and effectiveness of the proposed optimizations (§5.2.3).

Experimental testbed. To evaluate a realistic environment for container deployments [3], we evaluated the performance benchmarks using m4.xlarge virtual machine instances on Amazon EC2. The machine type has two cores of Intel Xeon E5-2686 CPU (4 hardware threads) assigned and 16GB RAM. The Linux kernel version was 4.14.13. For storage, we used a 100GB EBS volume of type GP2 formatted with ext4 filesystem mounted with default options. GP2 is an SSD-backed storage and attached via a dedicated network to the VM.

Benchmark: Phoronix suite. For the performance measurement, we used the disk benchmarks [39] from the Phoronix suite [18]. Phoronix is a meta benchmark that has a wide range of common filesystem benchmarks, applications, and realistic workloads. We compiled the benchmarks with GCC 6.4 and CNTR with Rust 1.23.0.

Methodology. For the performance comparison, we ran the benchmark suite once on the native filesystem (the baseline measurement) and compared the performance when we access the same filesystem through CNTRFS. The Phoronix benchmark suite runs each benchmark at least three times and automatically adds additional trials if the variance is too high. To compute the relative overheads with respect to the baseline, we computed the ratio between the native filesystem access and CNTRFS (*native/cntr*) for benchmarks where higher values are better (e.g. throughput), and the inverse ratio

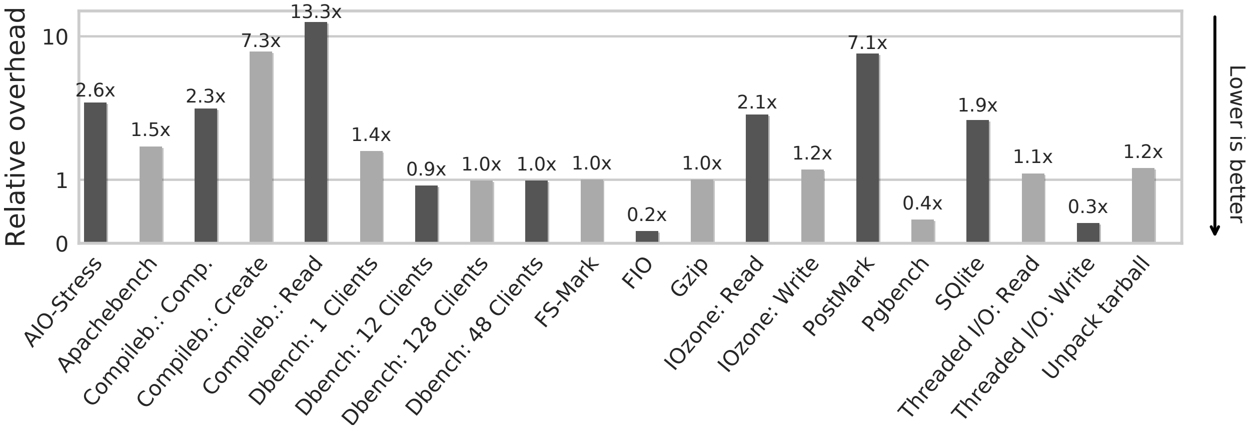


Figure 2: Relative performance overheads of CNTR for the `Phoronix` suite. The absolute values for each benchmark is available online on the openbenchmark platform [31].

(*cntr*/*native*), where lower values are better (e.g. time required to complete the benchmark).

5.2.1 Performance Overheads

We first present the summarized results for the entire benchmark suite. Thereafter, we present a detailed analysis of each benchmark individually (§5.2.2).

Summary of the results. Figure 2 shows the relative performance overheads for all benchmarks in the `Phoronix` test suite. We have made the absolute numbers available for each benchmark on the openbenchmark platform [31].

Our experiment shows that 13 out of 20 (65%) benchmarks incur moderate overheads below 1.5× compared to the native case. In particular, three benchmarks showed significantly higher overheads, including `compilebench-create` (7.3×) and `compilebench-read` (13.3×) and the `postmark` benchmark (7.1×). Lastly, we also had three benchmarks, where CNTRFS was faster than the native baseline execution: `FIO` (0.2×), `PostgreSQL Bench` (0.4×) and the write workload of `Threaded I/O` (0.3×).

To summarize, the results show the strengths and weaknesses of CNTRFS for different applications and under different workloads. At a high-level, we found that the performance of inode lookups and the double buffering in the page cache are the main performance bottlenecks in our design (much like they are for FUSE). Overall, the performance overhead of CNTR is reasonable. Importantly, note that while reporting performance numbers, we resort to the worst-case scenario for CNTR, where the "slim" application container aggressively uses the "fat" container to run an I/O-intensive benchmark suite. However, we must emphasize the primary goal of CNTR: to support auxiliary tools in uncommon operational use-cases, such as debugging or manual inspection, which are not dominated by high I/O-intensive workloads.

5.2.2 Detailed Experimental Results

We next detail the results for each benchmark.

AIO-Stress. `AIO-Stress` submits 2GB of asynchronous write requests. In theory, CNTRFS supports asynchronous requests, but only when the filesystem operates in the direct I/O mode. However, the direct I/O mode in CNTRFS restricts the `mmap` system call, which is required by executables. Therefore, all requests are, in fact, processed synchronously resulting in 2.6× slowdown.

Apache Web server. The `Apache Web server` benchmark issues 100K `http` requests for test files (average size of 3KB), where we noticed a slowdown of up to 1.5×. However, the bottleneck was not due to serving the actual content, but due to writing of the webserver access log, which triggers small writes (< 100 bytes) for each request. These small requests trigger lookups in CNTRFS of the extended attributes `security.capabilities`, since the kernel currently neither caches such attributes nor it provides an option for caching them.

Compilebench. `Compilebench` simulates different stages in the compilation process of the Linux kernel. There are three variants of the benchmark: (a) the `compile` stage compiles a kernel module, (b) the `read tree` stage reads a source tree recursively, and lastly, (c) the `initial creation` stage simulates a tarball unpack. In our experiments, `Compilebench` has the highest overhead of all benchmarks with the `read tree` stage being the slowest (13.4×). This is due to the fact that inode lookups in CNTRFS are slower compared to the native filesystem: for every lookup, we need one `open()` system call to get a file handle to the inode, followed by a `stat()` system call to check if we already have lookup-ed this inode in a different path due hardlinks. Usually, after the first lookup, this information can be cached in the kernel, but in this benchmark for every run, a different source tree with many files are read. The slowdown of

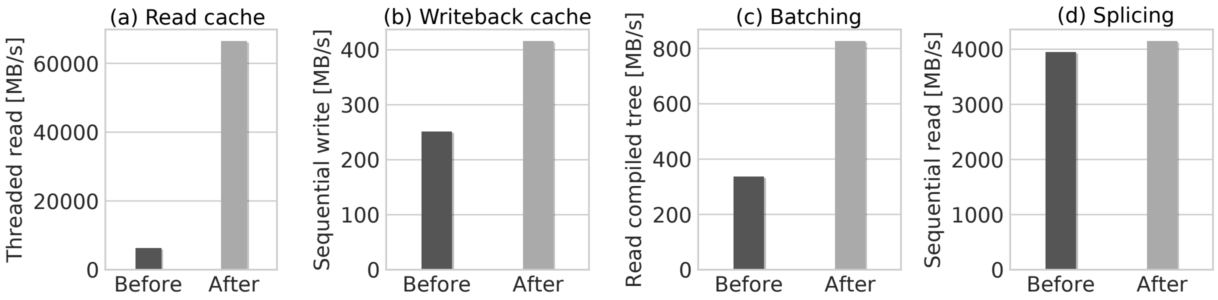


Figure 3: Effectiveness of optimizations

(a) `Threaded I/O` bench - Read (b) `IOZone` - Sequential write with 4GB (record size 4KB)
(c) `Compilebench` - Read (d) `IOZone` - Sequential read 4GB (record size 4KB)

lookups for the other two variants, namely the `compile stage` (2.3×) and `initial create` (7.3×) is lower, since they are shadowed by write operations.

Dbench. `Dbench` simulates a file server workload, and it also simulates clients reading files and directories with increasing concurrency. In this benchmark, we noticed that with increasing number of clients, CNTRFS is able to cache directories and file contents in the kernel. Therefore, CNTRFS does not incur performance overhead over the native baseline.

FS-Mark. `FS-Mark` sequentially creates 1000 1MB files. Since the write requests are reasonably large (16 KB per `write` call) and the workload is mostly disk bound. Therefore, there is no difference between CNTRFS and `ext4`.

FIO benchmark. The `FIO` benchmark profiles a fileserver and measures the read/write bandwidth, where it issues 80% random reads and 20% random writes for 4GB data with an average blocksize of 140KB. For this benchmark, CNTRFS outperforms the native filesystem by a factor of 4× since the writeback cache leads to fewer and larger writes to the disk compared to the underlying filesystem.

Gzip benchmark. The `Gzip` benchmark reads a 2GB file containing only zeros and writes the compressed version of it back to the disk. Even though the file is highly compressible, `gzip` compresses the file slower than the data access in CNTRFS or `ext4`. Therefore, there was no significant performance difference between CNTR and the native version.

IOZone benchmark. `IOZone` performs sequential writes followed by sequential reads of a blocksize of 4KB. For the write requests, as in the `apache` benchmark, CNTR incurs low overhead (1.2×) due to extended attribute lookup overheads. Whereas, for the sequential read request, both filesystems (underlying native filesystem and CNTRFS) can mostly serve the request from the page cache. For smaller read sizes (4GB) the read throughput is comparable for both CNTRFS and `ext4` filesystems because the data fits in the page cache. However, a larger workload (8GB) no longer fits into the page cache of CNTRFS and degrades the throughput significantly.

Postmark mailserver benchmark. `Postmark` simulates a mail server that randomly reads, appends, creates or deletes small files. In this benchmark, we observed higher overhead (7.1×) for CNTR. In this case, inode lookups in CNTRFS dominated over the actual I/O because the files were deleted even before they were sync-ed to the disk.

PGBench – PostgreSQL Database Server. `PGBench` is based on the `PostgreSQL` database server. It simulates both read and writes under normal database load. Like `FIO`, CNTRFS was faster in this benchmark also, since `PGBench` flushes the writeback buffer less often.

SQLite benchmark. The `SQlite` benchmark measures the time needed to insert 1000 rows in a SQL table. We observed a reasonable overhead (1.9×) for CNTR, since each insertion is followed by a filesystem `sync`, which means that we cannot make efficient use of our disk cache.

Threaded I/O benchmark. The `Threaded I/O` benchmark separately measures the throughput of multiple concurrent readers and writers to a 64MB file. We observed good performance for reads (1.1×) and even better performance for writes (0.3×). This is due to the fact that the reads can be mostly served from the page cache, and for the writes, our writeback buffer in the kernel holds the data longer than the underlying filesystem.

Linux Tarball workload. The `Linux tarball` workload unpacks the kernel source code tree from a compressed tarball. This workload is similar to the `create` stage of the `compilebench` benchmark. However, since the source is read from a single tarball instead of copying an already unpacked directory, there are fewer lookups performed in CNTRFS. Therefore, we incur relatively lower overhead (1.2×) even though many small files are created in the unpacking process.

5.2.3 Effectiveness of Optimizations

We next evaluate the effectiveness of the proposed optimizations in CNTR (as described in §3.3).

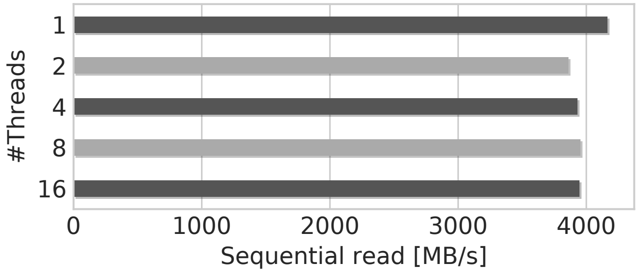


Figure 4: Multithreading optimization with IOZone: Sequential read 500MB/4KB record size with increasing number of CNTRFS threads.

Read cache. The goal of this optimization is to allow the kernel to cache pages across multiple processes. Figure 3 (a) shows the effectiveness of the proposed optimization for `FOPEN_KEEP_CACHE`: we observed 10× higher throughput with `FOPEN_KEEP_CACHE` for concurrent reads with 4 threads for the `Threaded I/O` benchmark.

Writeback cache. The writeback optimization was designed to reduce the amount of write requests by maintaining a kernel-based write cache. Figure 3 (b) shows the effectiveness of the optimization: CNTR can achieve 65% more write throughput with the writeback cache enabled compared to the native I/O performance for sequential writes for the `IOZone` benchmark.

Multithreading. We made CNTRFS multi-threaded to improve responsiveness when the filesystem operations block. While threads improve the responsiveness, their presence hurts throughput as measured in Figure 4 (up to 8% for sequential read in `IOZone`). However, we still require multithreading to cope with blocking filesystem operations.

Batching. To improve the directory and inode lookups, we batched requests to kernel by specifying the `PARALLEL_DIROPS` flag. We observed a speedup of 2.5× in the `compilebench` read benchmark with this optimization (Figure 3 (c)).

Splice read. Instead of copying memory into userspace, we move the file content with the `splice()` syscall in the kernel to achieve zero-copy I/O. Unfortunately, we did not notice any significant performance improvement with the splice read optimization. For instance, the sequential read throughput in `IOZone` improved slightly by just 5% as shown in Figure 3 (d).

5.3 Effectiveness of CNTR

To evaluate the effectiveness of CNTR's approach to reducing the image sizes, we used a tool called Docker Slim [11].

Docker Slim applies static and dynamic analyses to build a smaller-sized container image that only contains the files that are actually required by the application. Under the hood, Docker Slim records all files that have been accessed during a container run in an efficient way using the `fanotify` kernel module.

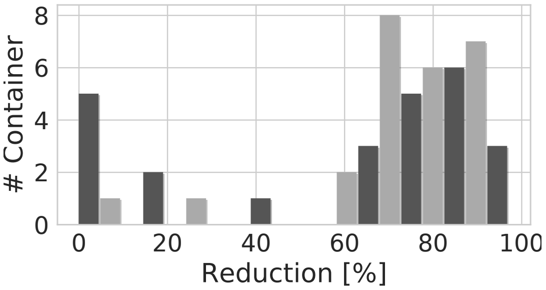


Figure 5: Reduction of container size after applying docker-slim on Top-50 Docker Hub images.

For our analysis, we extended Docker Slim to support container links, which are extensively used for service discovery and it is available as a fork [28].

Dataset: Docker Hub container images. For our analysis, we chose the Top-50 popular official container images hosted on Docker Hub [10]. These images are maintained by Docker and contain commonly used applications such as web servers, databases and web applications. For each image, Docker provides different variants of Linux distributions as the base image. We used the variant set to be `default` as specified by the developer.

Note that Docker Hub also hosts container images that are not meant to be used directly for deploying applications, but they are meant to be used as base images to build applications (such as language SDKs or Linux distributions). Since CNTR targets concrete containerized applications, we did not include such base images in our evaluation.

Methodology. For our analysis, we instrumented the Docker container with Docker Slim and manually ran the application so it would load all the required files. Thereafter, we build new smaller containers using Docker Slim. These new smaller images are equivalent to containers that developers could have created when having access to CNTR. We envision the developers will be using a combination of CNTR and tools such as Docker Slim to create smaller container images. Lastly, we tested to validate that the smaller containers still provide the same functionality.

Experimental results. On average, we could reduce the size by 66.6% for the Top-50 Docker images. Figure 5 depicts the histogram plot showcasing percentage of container size that could be removed in this process. For over 75% of all containers, the reduction in size was between 60% and 97%. Beside the applications, these containers are packaged with common used command line auxiliary tools, such as coreutils, shells, and package managers. For only 6 out of 50 (12%) containers, the reduction was below 10%. We inspected these 6 images and found out they contain only single executables written in Go and a few configuration files.

6 Related Work

In this section, we survey the related work in the space of lightweight virtualization.

Lambda functions. Since the introduction of AWS Lambda [1], all major cloud computing providers offer serverless computing, including Google Cloud Functions [15], Microsoft Azure Functions [5], IBM OpenWhisk [20]. Moreover, there exists a research implementation called Open Lambda [55]. In particular, serverless computing offers a small language runtime rather than the full-blown container image. Unfortunately, lambdas offer limited or no support for interactive debugging or profiling purposes [63] because the clients have no access to the lambda's container or container-management system. In contrast, the goal of the CNTR is to aim for lightweight containers, in the same spirit of lambda functions, but to also provide an on-demand mechanism for auxiliary tools for debugging, profiling, etc. As a future work, we plan to support auxiliary tools for lambda functions [43] using CNTR.

Microkernels. The microkernel architecture [54, 46, 56] shares a lot of commonalities with the CNTR architecture, where the applications/services are horizontally partitioned and the communication happens via the inter-process communication (IPC) mechanism. In CNTR, the application container obtains additional service by communicating with the "fat" container via IPC using CNTRFS.

Containers. Recently, there has been a lot of interest in reducing the size of containers, but still allowing access to the rich set of auxiliary tools. For instance, Toolbox [35] in CoreOS [7] allows to bind the mount of the host filesystem in a container to administrate or debug the host system with installing the tools inside the container. In contrast to Toolbox, CNTR allows bidirectional access with the debug container. Likewise, `nsenter` [27] allows entering into existing container namespaces, and spawning a process into a new set of namespaces. However, `nsenter` only covers namespaces, and it does not provide the rich set of filesystem APIs as provided by CNTR. Lastly, Slacker [53] proposed an opportunistic model to pull images from registries to reduce the startup times. In particular, Slacker can skip downloading files that are never requested by the filesystem. Interestingly, one could also use Slacker to add auxiliary tools such as gdb to the container in an "on-demand" fashion. However, Slacker could support additional auxiliary tools to a container, but these tools would be only downloaded to the container host, if the container is started by the user. Furthermore, Slacker also has a longer build time and greater storage requirements in the registry. In contrast, CNTR offers a generic lightweight model for the additional auxiliary tools.

Virtual machines. Virtual machines [25, 47, 51] provide stronger isolation compared to containers by running applications and the OS as a single unit. On the downside, full-fledged VMs are not scalable and resource-efficient [62]. To strike a balance between the advantages of containers and virtual machines, Intel Clear Containers (or Kata Containers) [21] and SCONE [45] offer stronger security properties for containers by leveraging Intel VT and Intel SGX, respectively. Likewise, LightVM [59] uses unikernel and optimized Xen to offer lightweight VMs. In a similar vein, CNTR allows creating lightweight containers, which are extensively used in the data center environment.

Unikernels and Library OSes. Unikernels [57, 58] leverage library OSes [61, 49, 65, 48] to selectively include only those OS components required to make an application work in a single address space. Unikernels use a fraction of the resources required compared to full, multipurpose operating systems. However, Unikernels also face a similar challenge as containers — If Unikernels need additional auxiliary tools, they must be statically linked in the final image as part of the library OS. Moreover, unikernel approach is orthogonal since it targets the kernel overhead, whereas CNTR targets the tools overhead.

7 Conclusion

We presented CNTR, a system for building and deploying lightweight OS containers. CNTR partitions existing containers into two parts: "slim" (application) and "fat" (additional tools). CNTR efficiently enables the application container to dynamically expand with additional tools in an on-demand fashion at runtime. Further, CNTR enables a set of new development workflows with containers:

- When testing the configuration changes, instead of rebuilding containers from scratch, the developers can use their favorite editor to edit files in place and reload the service.
- Debugging tools no longer have to be manually installed in the application container, but can be placed in separate debug images for debugging or profiling in production.

To the best of our knowledge, CNTR is the first generic and complete system that allows attaching to container and inheriting all its sandbox properties. We have used CNTR to debug existing container engines [32]. In our evaluation, we have extensively tested the completeness, performance, and effectiveness properties of CNTR. We plan to further extend our evaluation to include the nested container design.

Software availability. We have made CNTR along with the complete experimental setup publicly available [6].

Acknowledgments. We thank our shepherd Swaminathan Sundararaman and the anonymous reviewers for their helpful comments. The work is supported in part by the Alan Turing Institute and an Amazon Web Services Education Grant.

References

[1] Amazon AWS Lambdas. https://aws.amazon.com/lambda/.

[2] Amazon Elastic Container Service (ECS). https://aws.amazon.com/ecs/.

[3] Amazon's documentation on EBS volume types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html.

[4] Azure Container Service (AKS). https://azure.microsoft.com/en-gb/services/container-service/.

[5] Azure Functions. https://azure.microsoft.com/en-gb/services/functions/.

[6] Cntr homepage. https://github.com/Mic92/cntr.

[7] Container optimized Linux distribution. https://coreos.com/.

[8] CoreOS. https://coreos.com/.

[9] Docker. https://www.docker.com/.

[10] Docker repositories. https://hub.docker.com/explore/.

[11] Docker Slim. https://github.com/docker-slim/docker-slim.

[12] Docker Swarm. https://www.docker.com/products/docker-swarm.

[13] Docker switch to Alpine Linux. https://news.ycombinator.com/item?id=11000827.

[14] File system regression test on linux implemented for all major filesystems. https://kernel.googlesource.com/pub/scm/fs/ext2/xfstests-bld/+/HEAD/Documentation/what-is-xfstests.md.

[15] Google Cloud Functions. https://cloud.google.com/functions/.

[16] Google Compute Cloud Containers. https://cloud.google.com/compute/docs/containers/.

[17] Google: 'EVERYTHING at Google runs in a container'. https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/.

[18] Homepage of Phoronix test suite. https://www.phoronix-test-suite.com/.

[19] Homepage of SELinux. https://selinuxproject.org/page/Main_Page.

[20] IBM OpenWhisk. https://www.ibm.com/cloud/functions.

[21] Intel Clear Containers. https://clearlinux.org/containers.

[22] Kubernetes. https://kubernetes.io/.

[23] Lightweight standard libc implementation. https://www.musl-libc.org/.

[24] Linux Containers. https://linuxcontainers.org/.

[25] Linux Kernel Virtual Machine (KVM). https://www.linux-kvm.org/page/Main_Page.

[26] Manual of AppArmor. http://manpages.ubuntu.com/manpages/xenial/en/man7/apparmor.7.html.

[27] nsenter. https://github.com/jpetazzo/nsenter.

[28] Our fork of Docker Slim used for evaluation. https://github.com/Mic92/docker-slim/tree/cntr-eval.

[29] Our fork the nix rust library. https://github.com/Mic92/cntr-nix.

[30] RancherOS. https://rancher.com/rancher-os/.

[31] Raw benchmark report generated by phoronix test suite. https://openbenchmarking.org/result/1802024-AL-CNTREVALU05.

[32] Root cause analysis in unprivileged nspawn container with cntr. https://github.com/systemd/systemd/issues/6244#issuecomment-356029742.

[33] Rust library for filesystems in userspace. https://github.com/zargony/rust-fuse.

[34] Rust library that wraps around the Linux/Posix API. https://github.com/nix-rust/nix.

[35] Toolbox. https://github.com/coreos/toolbox.

[36] Twelve-Factor App. https://12factor.net/config.

[37] Website of the lxc container engine. https://linuxcontainers.org/.

[38] Website of the rkt container engine. `https://coreos.com/rkt/`.

[39] Wiki page for the Phoronix disk test suite. `https://openbenchmarking.org/suite/pts/disk`.

[40] *namespaces(7) Linux User's Manual*, July 2016.

[41] *cgroups(7) Linux User's Manual*, September 2017.

[42] *mount(8) Linux User's Manual*, September 2017.

[43] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[44] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt. Improving Docker Registry Design based on Production Workload Analysis. In *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[45] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[46] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[47] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[48] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013.

[49] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[50] P. Bhatotia, A. Wieder, R. Rodrigues, F. Junqueira, and B. Reed. Reliable Data-center Scale Computations. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.

[51] R. J. Creasy. The Origin of the VM/370 Time-sharing System. *IBM J. Res. Dev.*, 1981.

[52] L. Du, R. Y. Tianyu Wo, and C. Hu. Cider: A Rapid Docker Container Deployment System through Sharing Network Storage. In *Proceedings of the 19th International Conference on High Performance Computing and Communications (HPCC)*, 2017.

[53] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[54] G. Heiser and K. Elphinstone. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Transaction of Computer Systems (TOCS)*, 2016.

[55] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2016.

[56] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[57] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[58] A. Madhavapeddy and D. J. Scott. Unikernels: The Rise of the Virtual Library Operating System. *Communication of ACM (CACM)*, 2014.

[59] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[60] D. P. Quigley, J. Sipek, C. P. Wright, and E. Zadok. UnionFS: User- and Community-oriented Development of a Unification Filesystem. In *Proceedings of the 2006 Linux Symposium (OLS)*, 2006.

[61] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[62] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference (Middleware)*, 2016.

[63] J. Thalheim, P. Bhatotia, and C. Fetzer. INSPECTOR: Data Provenance Using Intel Processor Trace (PT). In *IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[64] J. Thalheim, A. Rodrigues, I. E. Akkus, R. C. Pramod Bhatotia, B. Viswanath, L. Jiao, and C. Fetzer. Sieve: Actionable Insights from Monitored Metrics in Distributed Systems. In *Proceedings of Middleware Conference (Middleware)*, 2017.

[65] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014.

[66] B. K. R. Vangoor, V. Tarasov, and E. Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

Throwhammer: Rowhammer Attacks over the Network and Defenses

Andrei Tatar
VU Amsterdam

Radhesh Krishnan Konoth
VU Amsterdam

Elias Athanasopoulos
University of Cyprus

Cristiano Giuffrida
VU Amsterdam

Herbert Bos
VU Amsterdam

Kaveh Razavi
VU Amsterdam

Abstract

Increasingly sophisticated Rowhammer exploits allow an attacker that can execute code on a vulnerable system to escalate privileges and compromise browsers, clouds, and mobile systems. In all these attacks, the common assumption is that attackers first need to obtain code execution on the victim machine to be able to exploit Rowhammer either by having (unprivileged) code execution on the victim machine or by luring the victim to a website that employs a malicious JavaScript application. In this paper, we revisit this assumption and show that an attacker can trigger and exploit Rowhammer bit flips directly from a remote machine by *only* sending network packets. This is made possible by increasingly fast, RDMA-enabled networks, which are in wide use in clouds and data centers. To demonstrate the new threat, we show how a malicious client can exploit Rowhammer bit flips to gain code execution on a remote key-value server application. To counter this threat, we propose protecting unmodified applications with a new buffer allocator that is capable of fine-grained memory isolation in the DRAM address space. Using two real-world applications, we show that this defense is practical, self-contained, and can efficiently stop remote Rowhammer attacks by surgically isolating memory buffers that are exposed to untrusted network input.

1 Introduction

A string of recent papers demonstrated that the Rowhammer hardware vulnerability poses a growing threat to system security. From a potential security hole in 2014 [36], it grew into an attack vector to mount end-to-end exploits in browsers [15, 29, 52], cloud environments [47, 51, 60], and smartphones [24, 59]. Recent work even generated Rowhammer-like bit flips on flash storage [17, 38]. Even so, however advanced the attacks have become and however worrying for the research community, these attacks never progressed beyond local privilege escalations or sandbox escapes. The attacker needs the ability to run code on the victim machine in order to flip bits in sensitive data. Hence, Rowhammer posed little threat from attackers without code execution on the victim machines. In this paper, we show that this is no longer true and attackers can flip bits by only sending network packets to a victim machine connected to RDMA-enabled networks commonly used in clouds and data centers [1, 20, 45, 62].

Rowhammer exploits today Rowhammer allows attackers to flip a bit in one physical memory location by aggressively reading (or writing) other locations (i.e., *hammering*). As bit flips occur at the physical level, they are beyond the control of the operating system and may well cross security domains. A Rowhammer attack requires the ability to hammer memory sufficiently fast to trigger bit flips in the victim. Doing so is not always trivial as several levels of caches in the memory hierarchy often absorb most of the memory requests. To address this hurdle, attackers resort to accessing cache eviction buffers [12] or using direct memory access (DMA) [59] for hammering. But even with these techniques in place, triggering a bit flip still requires hundreds of thousands of memory accesses to specific DRAM locations within tens of milliseconds. As a result, the current assumption is that Rowhammer may only serve local privilege escalation, but not to launch attacks from over the network.

Remote Rowhammer attacks In this paper, we revisit this assumption. While it is true that millions of DRAM accesses per second is harder to accomplish from across the network than from code executing locally, today's networks are becoming very fast. Modern NICs are able to transfer large amounts of network traffic to remote memory. In our experimental setup, we observed bit flips when accessing memory 560,000 times in 64 ms, which translates to 9 million accesses per second. Even regular 10 Gbps Ethernet cards can easily send 9 million packets per second to a remote host that end up being stored on the host's memory. Might this be enough for an attacker to effect a Rowhammer attack from across the network? In the remainder of this paper, we demonstrate that this is the case and attackers can use these bit flips induced by network traffic to compromise a *remote server* application. To our knowledge, this is the first reported case of a Rowhammer attack over the network. Specifically, we managed to flip bits remotely using a commodity 10 Gbps network. We rely on the commonly-deployed RDMA technology in clouds and data centers for reading from remote DMA buffers quickly to cause Rowhammer corruptions outside these untrusted buffers. These corruptions allow us to compromise a remote memcached server without relying on any software bug.

Mitigating remote Rowhammer attacks It is unclear whether existing hardware mitigations can protect against these dangerous network attacks. For instance, while clouds and data centers may (and sometimes do) use ECC memory to guard against bit flips, researchers have, from the first paper on Rowhammer [36], warned that ECC may not be sufficient to protect against such attacks. Targeted Row Refresh, specifically designed to address Rowhammer is also shown not to be always effective [41, 59]. Unfortunately, existing software defenses are also unprepared to deal with network attacks: ANVIL [12] relies on performance counters that are not available for DMA, CATT [16] only protects the kernel from user-space attacks and VUsion [47] only protects the memory deduplication subsystem.

We make the observation that compared to local attackers, remote attackers can only target memory that is allocated for DMA buffers. Hence, instead of protecting the entire memory, we only need to make sure that these buffers cannot cause bit flips in the rest of the system. Specifically, we show that we can isolate the buffers for fast network communication using a new memory allocation strategy that places CATT-like guard zones around them. These guard zones absorb any attacker-generated bit flips, protecting the rest of the system. Properly implementing this allocation strategy is not trivial: the guard zones need to be placed in the DRAM address space to effectively absorb the bit flips. Unfortunately, the physical address space is *not* consecutively mapped to the DRAM address-space, unlike what is assumed in existing defenses [12, 16]. Memory controllers use complex functions to translate a physical address into a DRAM address. We therefore present a new allocator, called ALIS (ALlocations ISolated), which uses a novel approach to translate between physical address-space and DRAM address-space and safely allocates the DMA buffers and their guard zones. Since we need to protect only a limited number of DMA buffers, doing so is inexpensive as we show using microbenchmarks and two real-world applications.

Contributions We make the following contributions:

- We describe Throwhammer, the first Rowhammer profiling tool that scans a host over the network for bit flips. We evaluate Throwhammer using different configurations (i.e., different link speeds and DIMMs). We then show how an attacker can use these bit flips to exploit a remote memcached server.

- We design and implement ALIS, a new allocator for safe allocation of network buffers. We show that ALIS correctly finds guard rows at the DRAM address space level, provided an address mapping that satisfies certain prerequisites. Furthermore, we show that ALIS is compatible with existing software. We further evaluate ALIS using microbenchmarks and real-world applications to show that it incurs negligible performance and memory overhead.

- We release Throwhammer and ALIS as open-source software in the URL that follows.

  `https://vusec.net/projects/throwhammer`

  Concerned parties can use Throwhammer to check for remote bit flips and ALIS for protecting their applications against remote Rowhammer attacks.

2 Background

With software becoming increasingly more difficult due to a variety of defenses deployed in practice [11, 25, 30, 39, 55, 57], security researchers have started exploring new directions for exploitation. For example, CPU caches can be abused to leak information [19, 26, 37, 42, 48, 61] or wide-spread reliability issues in hardware [18, 36] can be abused to compromise software [29, 52, 59]. These attacks require code execution on the victim machine. In this paper, we show for the first time that this requirement is not strictly necessary, and it is possible to trigger reliability issues in DRAM by merely sending network packets. We provide necessary background on DRAM and its unreliabilities and high-speed networks that expose them remotely.

2.1 Unreliable DRAM

DRAM Organization The basic unit of storage in DRAM is cell made out of a capacitor used to hold the value of a single bit of information. Since capacitors leak charge over time, the memory controller should frequently (typically every 64 ms) recharge them in order to maintain the stored values, a process known as *refreshing*. DRAM cells are ganged together to form what is known as a *row* (typically 1024 cells or *columns* wide).

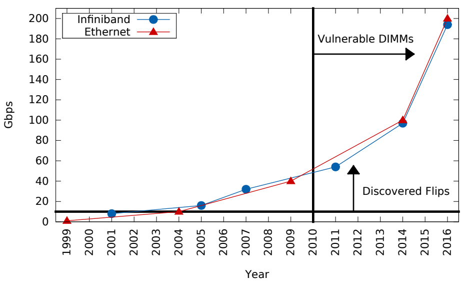


Figure 1: Trends in network performance and Rowhammer.

Whenever a row is accessed the contents of that particular row are put on a special buffer, called *row buffer*, and the row is said to be activated. Once access is finished, the activated row is written (i.e., recharged) with the contents of the row buffer. Multiple rows along with a row buffer are stacked together to form a *bank*. There are multiple banks on a DRAM integrated circuit (IC). Multiple DRAM ICs are laid out to for a DRAM *rank*. The DRAM chips are accessed in parallel when reading a memory word. For example, with a DIMM that has 8 bit wide ICs, eight ICs are accessed in parallel to form a 64 bit memory word.

DRAM addressing Addressing a memory word within a DRAM rank is done by the system memory controller using three addresses: bank, row and column. Further parallelism can be added by having two ranks on a single memory module (DIMM), adding multiple DIMMs on the same memory bus (also known as *channel*), and providing multiple independent memory channels. Hence, to address a specific word of memory, the memory controller uses a <channel, DIMM, rank, bank, row, column> hextuple. This hextuple, which we call a *DRAM address* is constructed from the physical memory address bits using formulas which are either documented [10] or have been (partially) reverse engineered [49]. An important take-away here is that contiguous physical address space is not necessarily contiguous in the DRAM address space where Rowhammer bit flips happen. This information is important when developing our defense discussed in §6.

Rowhammer As DRAM chips become denser, the charge used for each capacitor to denote the two bit states is reduced. A reduced charge level increases the possibility of errors. Kim et al. [36] show that intentionally activating a row many times in a short duration (i.e., Rowhammering) can cause the charge in the capacitors to leak in close-by rows. If this happens fast enough, before the memory controller can refresh the adjacent rows, this charge leakage passes a certain threshold and as a result bits in these adjacent, or *victim*, rows will flip. To exploit these flips, the attackers need to first find bit flips in interesting offsets within a memory page and then force the system to store sensitive information on that memory page. For instance, the first known exploit by Seaborn [52] finds a memory page with a bit flip that can affect page table entries. It then frees that memory page and sprays the system with page table pages. The hope is that the page that is now freed is used by a page table page and the bit flip causes the page table entry to point to another page table page, effectively giving the attacker control over all of physical memory. Similarly, Rowhammer.js [29], Drammer [59] and Xiao et al. [60] target page table pages but focus on browser, mobile and cloud environments respectively. Other attacks target cryptographic keys [51] or JavaScript objects [15, 24].

2.2 Fast Networks

Figure 1 shows the evolution of network performance over time. Since 2010, the trend follows an exponential increase in the amount of available network bandwidth. This is putting a lot of pressure on other components of the system, namely the CPU and the memory subsystem, and has forced systems engineers to rethink their software stack in order to make use of all this bandwidth [22, 23, 33, 34, 43, 44].

Figure 1 also shows that DIMMs with the Rowhammer vulnerability have been produced since 2010 and their production continues to date [40, 59]. As we will show in §4, we observed bit flips with capacities available in 10 Gbps or faster networks, suggesting that already back in 2010, Rowhammer was exploitable over the network.

While faster than 10 Gbps networks are very common in data centers on bare metal [45, 53, 62], even today's clouds offer high-speed networking. Amazon EC2 provides VMs with 20 Gbps connectivity [2] and Microsoft Azure provides VMs with 56 Gbps [1]. As we will soon show 10 Gbps networks already make remote bit flips a dangerous threat to regular users today.

Remote DMA To achieve high-performance networking, some systems entirely remove the interruptions and expensive privilege switching from the fast path and deliver network packets directly to the applications [13, 31, 50]. Such approaches often resort to polling in order to guarantee high-performance, wasting CPU cycles that are becoming more precious as Moore's law stagnates and the available network bandwidth per core increases.

To reduce the load on the CPU, some networking equipment include the possibility for Remote Direct Memory Access (RDMA). Figure 2 compares what happens when a client application sends a packet in a normal network compared to one with RDMA support. Without RDMA, the client machine's CPU first needs to copy the packet's content (e.g., an HTTP request) from an application buffer to a DMA buffer. The client machine's oper-

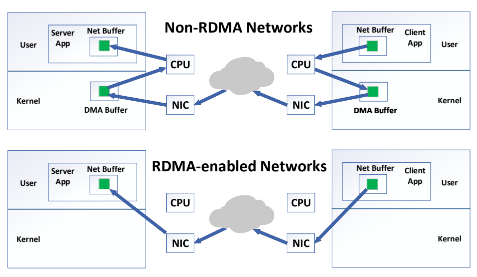


Figure 2: RDMA allows zero-copy network communication.

ating system then signals the NIC that the packet is ready for network transfer. The NIC then reads the packet using DMA and sends it over the wire. On the server side, the server's NIC receives the packet from the wire and copies it to a DMA buffer that is pre-configured to the NIC. It then signals the server's CPU that a packet has arrived. The CPU then copies the packet's content to the server application's buffer.

With RDMA, there is no need to involve the CPU on both client and server for packet transfer. The server and client applications both configure DMA buffers to the NIC through interfaces that are provided by the operating system. When the client application wants to send a packet to a server application, it directly writes it to its buffer. It then signals its NIC that the packet is ready for transfer. The NIC then sends the packet over the wire. On the server side, the NIC receives the packet and directly writes it to the buffer that has previously been configured by the server application. The server application can then be notified that a new packet has arrived or it can poll its own buffer. RDMA can boost existing protocols such as NFS [46] and new applications can use its functionalities to achieve better performance. Examples include databases [43], distributed hash tables and in-memory key-value stores [22, 23, 33, 34, 44].

RDMA's prevalence Data centers and cloud providers such as Google [45] and Microsoft [1, 62, 20] use RDMA to improve the performance of their clusters. Microsoft very recently announced RDMA support for SMB file sharing in the workstation edition of Windows [5], suggesting RDMA-enabled networks are spreading into the workstation market. Cloud providers are already selling virtual machines with RDMA support. For example, Microsoft Azure [3] and ProfitBricks [8] provide offerings with high-speed RDMA networks.

3 Threat Model

We consider attackers that generate and send legitimate traffic through a high-speed network to a target server. A common example is a client that sends requests to a cloud or data center machine that runs a server application. We assume that the target machine is vulnerable to Rowhammer bit flips [51, 60]. We further assume that the target system benefits from IOMMU protection. With IOMMU, the server's NIC is not allowed to write to memory pages that are not part of the pre-configured DMA areas by the server application. The end goal of the attacker is to bypass RDMA's security model by modifying bits outside of memory areas that are registered for DMA in order to compromise the system.

4 Bit Flipping with Network Packets

To investigate the possibility of triggering bit flips over the network, we built the first Rowhammer test tool that scans for bit flips by repeatedly sending or receiving packets to/from a remote machine, called Throwhammer. Throwhammer is implemented using 1831 lines of C code and runs entirely in user-space without requiring any special privileges. We will make this tool available so that interested parties can check for remote bit flips.

4.1 Throwhammer's Implementation

Throwhammer makes use of RDMA capabilities for transferring packets efficiently. Throwhammer has two components: a server and a client process running on two nodes connected via an RDMA network. On the server side, we allocate a large virtually-contiguous buffer and configure it as a DMA buffer to the NIC. We set all bits to one when checking for one-to-zero bit flips and do the reverse when checking for zero-to-one bit flips.

On the client side, we repeatedly ask the server's NIC to send us packets with data from various offsets within this buffer. Given the remote nature of our attack, we cannot make any assumption on the physical addresses that map our target DMA buffers and cannot rely on side channels for inferring this information [28]. Fortunately, on the server side, the Linux kernel automatically turns the memory backing our RDMA buffer into huge pages with its `khugepaged` daemon. This allows us to perform double-sided Rowhammer similar to Rowhammer.js [29] and Flip Feng Shui [51]. Periodically, we check the entire buffer at the server for bit flips.

To make the best possible use of the network capacity, we spawn multiple threads in Throwhammer. At each round, two *aggressor* addresses are chosen and all the threads send/receive packets that read from these two addresses for a pre-defined number of times. We make no effort in synchronization between these threads, so multiple network packets may hit the row buffer. While we leave potential optimizations for selecting aggressor addresses more carefully and better synchronization to fu-

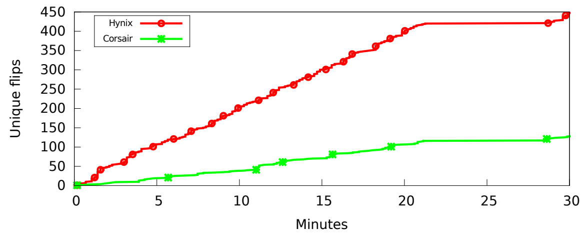


Figure 3: Number of unique Rowhammer bit flips over time using two sets of DIMMs over a 40 Gbps Ethernet network.

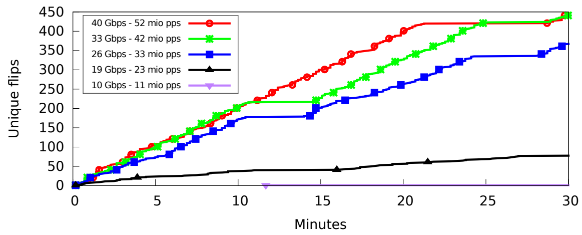


Figure 4: Number of unique Rowhammer bit flips on Hynix DIMMs over time using different network configurations.

ture work, we show that Throwhammer already can trigger bit flips in 10 Gbps networks and above.

4.2 Results

Testbed We use two machines each with 8-core Haswell i7-4790 processors connected using Mellanox ConnectX-4 single port NICs as our evaluation testbed. Note that these cards are already old: at the time of this paper's submission, two newer generations of these cards (ConnectX-5 and ConnectX-6) are available, but we show that it is already possible to trigger bit flips with our older generation cards. We experiment with different DIMMs and varying network performance.

DIMMs We chose two pairs of DDR3 DIMMs configured in dual-channel mode, one from Hynix and one from Corsair. These DIMMs already show bit flips when we run the open source Rowhammer test [6]. We configured our NICs in 40 Gbps mode and ran Throwhammer for 30 minutes. Figure 3 shows the number of unique bit flips as a function of time over these two sets of DIMMs on the server *triggered by transmitting packets*. We could flip 464 unique bits on the Hynix DIMMs and 185 unique bits on the Corsair DIMMs in 30 minutes. While these bit flips are already enough for exploitation, we believe that it is possible to trigger many more bit flips with a more optimized version of Throwhammer.

Network performance To understand how the network performance affects bit flips, we used the Hynix modules. We first configured our NICs in 10 Gbps Ethernet which can be saturated with 2 threads. We then configured our NICs in 40 Gbps Ethernet which can be saturated with 10 threads. The number of bit flips depends on the number of packets that we can send over the network (i.e., how many times we force a row to open) rather than the available bandwidth. For example, to trigger a bit flip that happens by reading 300,000 times from each aggressor address in the refresh window of 64 ms locally, in perfect conditions (e.g., proper synchronization) we need to be able to transmit $\frac{1000}{64} \times 300,000 \times 2 = 9.375$ million packets per second (pps). Unfortunately our NICs do not provide an option to reduce the bandwidth (or pps for that matter) in between 40 Gbps and 10 Gbps. We can however use fewer threads to emulate what the number of bit flips in networks that provide a bandwidth between 10 Gbps and 40 Gbps (e.g., Amazon EC2 [2]). We measure the pps for each configuration and use it to extrapolate the network bandwidth. Figure 4 shows the number of unique bit flips in different configurations as a function of time. With 10 Gbps, we managed to trigger one bit flip after 700.7 seconds, showing that commodity networks found in companies or university LANs are fast enough for triggering bit flips by transmitting network packets. Starting with faster networks than 10 Gbps, Throwhammer can trigger many more bit flips during the 30 minutes window. Again, we believe a more optimized version of Throwhammer can potentially generate more bit flips, especially on 10 Gbps networks.

5 Exploiting Bit Flips over the Network

We now discuss how one can exploit remote bit flips caused by accessing RDMA buffers quickly. The exploitation is similar to local Rowhammer exploits: the attacker needs to force the system to store sensitive data in vulnerable memory locations before triggering Rowhammer to corrupt the data in order to compromise the system. We exemplify this by building an end-to-end exploit against RDMA-memcached, a key-value store with RDMA support [32].

5.1 Memcached architecture

Memcached stores key/value pairs as *items* within memory *slabs* of various sizes. By default, the smallest slab class size is 96 bytes, with the largest being 1 MB. Memory allocated is broken up into 1 MB sized chunks and then assigned into slab classes as necessary. For rapid retrieval of keys, memcached uses a hash table, with colliding items chained in a singly-linked list.

The main data structure used for storing key/value *items* is called `struct _stritem`, as shown in Figure 5a. The first 50 bytes are used to store item metadata, while the remaining space is used to store the key and the value. Items are chained together into two lists. A

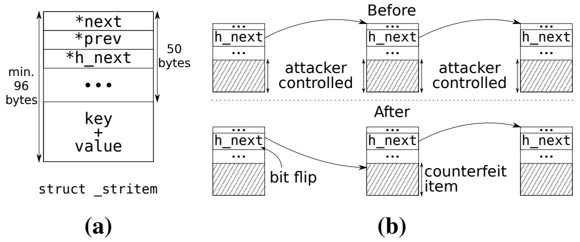


Figure 5: Memcached Exploit.

first doubly linked list (LRU, identified by the next and prev pointers) is updated during GET hits by relinking recent items at the head, and is traversed when freeing unused items. A second singly linked list (hash chain, identified by the h_next pointer) is traversed when looking up keys with colliding hashes. Another notable field is nbytes, the length of the value. *Slabs* come in fixed number of classes decided at compile-time with their metadata stored in a global data structure named slabclass. Crucially for our purposes, this data structure contains slots, a pointer to (a list of) "free" items which are used for subsequent SET operations.

5.2 Exploiting memcached

The attack progresses in four steps. In the first step, we search for bit flips using GET requests. Once we find an exploitable bit flip, we perform memory massaging [51] to land a target struct _stritem on the memory location with a bit flip. In the third step, we corrupt the hash chain to make a h_next value point to a counterfeit item that we encode inside a valid item. Our counterfeit item provides us with limited read and write primitives, using program logic triggered by GET requests. Finally, we target our limited write primitive towards the slabclass data structure, escalating it to arbitrary write and code execution. We describe each of the steps in more detail next.

Finding exploitable bit flips We spray the entire available memory with 1 MB sized key-value items with values made out of binary value one (when looking for 1 to 0 bit flips). Filling up all the available memory makes sure that some key-value items eventually border on the initial 16 MB RDMA buffers. Our experiments show that this is always the case. The attacker now remotely hammers the initial 16 MB RDMA buffers to trigger bit flips in the pages that belong to the adjacent rows where some of the 1 MB items are now stored. After that, the attacker reads back the items with GET requests to find out which bit offsets have been corrupted. We now discuss which offsets are exploitable.

Our target is corrupting the hash chain (i.e., the h_next pointer of a struct _stritem). As shown in Figure 5b, this allows us to pivot the h_next pointer to a counterfeit item that we encode inside a legitimate item — similar to Dedup Est Machina [15] and GLitch [24]. Assuming we can reuse a 1 MB item for smaller items, we have to see which size class we should pick for our target items so that one of the items' h_next pointer lands on a memory location with a bit flip. We do this analysis for every bit flip to see whether we can find a suitable size class for exploitation.

There are two challenges that we need to overcome for this strategy to work: first, we need to be able to chain many items together and second, we need to force memcached to reuse memory backing the 1 MB item with an exploitable bit flip for smaller items of the right size for corrupting their h_next pointer. We discuss how we overcome these challenges next.

Memory massaging We first need to craft memcached items with different keys which hash to the same value. Items with colliding keys make sure that the h_next pointer always points to an item that we control. Memcached uses the 32 bit Murmur3 hash function on the key to find the slot in a hashtable. This hash function is not cryptographically secure and we could easily generate millions of colliding 8 byte keys.

The simplest way to address the second challenge, is to issue a DELETE request on the target 1 MB item. We previously calculated the exact size class that would allow us to land an h_next pointer on a location with a bit flip. We can reassign the memory used by the deleted 1 MB item to the slab cache of the target item's size class using the slabs reassign command from the memcached client. Even without slabs reassign, we can easily trigger the reuse by just creating many items of the target size. The LRU juggler component in the memcached watches for free chunks in a slab class and reassigns any free ones to the global page pool.

While it is possible to deterministically reuse 1 MB items after an exploitable bit flip in a complete implementation of RDMA-memcached, the current version of RDMA-memcached does not support DELETE or slabs reassign. In these cases, we can simply spray the memory with items with a size that maximizes the of probability of corrupting the h_next pointer. We later report on the success rate of both attacks.

Corrupting the target item Once we land our target item on the desired location in memory, we re-trigger the bit flip by transmitting packets from the RDMA buffers. This causes the corruption of the target item's h_next pointer. With the right corruption, the pointer will now point inside another item whose key/value contents we control. By carefully crafting a counterfeit header inside the value area, we gain either a limited read or write capability. Issuing a GET request on our counterfeit item will now retrieve nbytes contiguous bytes from memcached's address space, provided the memory is mapped. Attempting to read unmapped memory is handled gracefully with an error being returned to the client, preventing unwanted crashes. If our counterfeit item is not LRU-linked (i.e., `next=prev=NULL`), the `GET` handler returns without any additional side-effects. A linked item, however, will trigger a relink operation on the next `GET`. By controlling the `next (n)` and `prev (p)` pointers and taking advantage of the unlink step, we gain a limited write primitive, where `*p=n` and `*(n+8)=p`, if `p` and `n` are respectively non-NULL.

Escalation The RDMA-memcached binary is not compiled as a Position Independent Executable (PIE); hence, we know the starting address of the `.data` and `.got` sections. Using this information, we point our write primitive at the `slabclass` data structure, aiming to overwrite the `slots` pointer of a particular class. We set up our counterfeit item with `next=&(slabclass[i].slots)-8`, where `i` is the slab class corresponding to our intended payload's size, and with `prev` to the target address of our choosing. The first `GET` operation on our counterfeit item will trigger the unlinking procedure which overwrites the `slots` pointer with `prev`, with the side effect of writing `next` to `*prev`. The next `SET` on our chosen class will store the new item in memory at the target address. Since we control the key and value of this new item, this gives us an arbitrary write primitive. We can further use our arbitrary write to corrupt the GOT to redirect the control flow and achieve code execution.

Results For the deterministic attack, we can successfully exploit 1.17% of 0 to 1 and 1.15% of 1 to 0 all possible bit flips. On the Hynix DIMMs, it takes us 5.1 minutes to find an exploitable bit flip and on the Corsair DIMMs it takes us 19.2 minutes to find an exploitable bit flip. With the non-deterministic attack, the best size class for spraying is items of size 384 bytes with 0.2% of the 0 to 1 bit flips resulting in a successful exploitation and 0.5% of them resulting in a crash. The results are similar with 1 to 0 bit flips.

6 Isolated Memory Allocation with ALIS

We now present an effective technique for defending against remote Rowhammer attacks using DRAM-aware allocation of network buffers. We first briefly discuss the main intuition behind our allocator, ALIS, before describing the associated challenges. We then show how ALIS overcomes these challenges for arbitrary physical-to-DRAM address mappings.

6.1 Challenges of Finding Adjacent Rows

The main idea behind ALIS is simple: given that Rowhammer bit flips happen in victim rows adjacent to aggressor rows, we need to make sure that all accessible rows in an isolated buffer are separated from the rest of system memory by at least one *guard row* used to absorb said bit flips. The implementation of this idea, however, is not simple because finding all possible victim rows along with their neighbors is not straightforward.

Given that bit flips happen on the DRAM ICs, ALIS should isolate rows in the DRAM address space. Recall from §2.1 that the DRAM address space is defined using the <channel, DIMM, rank, bank, row, column> hextuple. We use the term *row* to refer to memory locations addressed by <channel, DIMM, rank, bank, row, *>, where channel, DIMM, rank, bank and row are all fixed values. Given a singular row R at DRAM address <C, D, Ra, B, R, *> to be isolated, our aim is to allocate all DRAM addresses <C, D, Ra, B, R - 1, *> and <C, D, Ra, B, R + 1, *> (i.e., rows R - 1 and R + 1) as guard.

A common assumption made by both Rowhammer attacks [12, 15, 29, 51, 59] and defenses [12, 16] is that rows sharing the same row address (i.e., <*, *, *, *, row, *> memory locations) are contiguously mapped in physical memory. That is to say, while accessing physical memory addresses in an ascending order, the memory controller would activate the same numbered row across all its available banks, ranks, DIMMs and channels before moving on to the next. This assumption, however, does not hold for most real-world settings, since memory controllers have considerable freedom in translating physical addresses to DRAM addresses. One such example is presented in Figure 6a, which shows physical-to-DRAM address translation on an AMD CPU with 4 GB of single-rank memory [10]. Another example is shown in Figure 6b, with a non-linear physical address space to DRAM address space translation on an Intel Haswell CPU with dual-channel, dual-ranked memory with rank-mirroring [54]. As a result, existing attacks can become much more effective in finding bit flips if they take the translation between physical and DRAM address spaces into account. Similarly, current defenses only protect against bit flips caused by existing attacks that do not take this translation into account.

A correct solution must therefore be conservative in its assumptions about physical to DRAM address translation. In particular, we cannot assume that the contents of a DRAM row will be mapped to a contiguous area of physical memory. Similarly, we cannot assume that a physical page frame will be mapped to a single DRAM row. As an example of the latter, Figure 6c shows the physical to DRAM address space translation on an AMD CPU with channel-interleaving enabled by default [10].

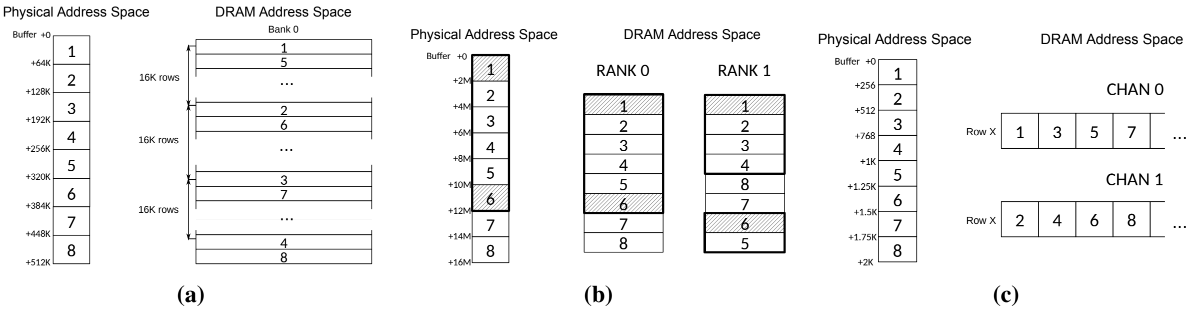


Figure 6: Examples of nonlinear DRAM address mappings taken from real systems [10, 49, 54].

6.2 Design

We now discuss how ALIS addresses these challenges. We define the *row group* of a page to be the set of rows that page maps onto. Similarly, the *page group* of a row is the set of pages with portions mapped to that row. In addition, in the context of a user-space process, we say a page is allocated if it is mapped by the system's MMU into its virtual address space. Likewise, we say a row is *full* if all pages in its page group are allocated, and *partial* if only a strict subset of these are allocated. If no page is allocated we call a row *empty*.

ALIS requires a physical to DRAM address mapping that satisfies the following condition: any two pages of an arbitrary row's page group have identical row groups themselves. If this condition holds, we can prove the following two properties:

(1) Enumeration property: To list all rows accessible by owning pages in a page group, it is sufficient to list any such page's row group.

(2) Fullness property: Rows that share page groups have the same allocation status — a row is full, partial or empty if and only if all rows in its pages' row group are respectively full, partial or empty.

For the interested reader we present a formal description of these properties, along with sufficiency criteria and proof that they hold for common architectures in [7].

Assuming a mapping where these properties hold, we now discuss how ALIS allocates isolated RDMA buffers.

6.3 Allocation Algorithm

Preparatory Steps. Initially, ALIS reserves a buffer and *locks* it in memory, so that any virtual memory mappings are not changed through the course of allocation or usage. Subsequently, ALIS translates the physical pages that back the reserved buffer into to their respective DRAM addresses. Translating from physical addresses to DRAM addresses is performed using mapping functions either available through manufacturer documentation [10] or previously reverse-engineered [49]. At the end of this step, we have a complete view of the allocated buffer in DRAM address space.

Pass 1: Marking. ALIS iterates through the rows of the buffer in DRAM address order and marks all (allocated) pages of a row as follows: Partially allocated rows have their pages marked as UNSAFE. Completely allocated (i.e. full) rows preceded or followed by a partially allocated or empty row are marked EDGE. Due to the fullness property we can be certain that any partial row groups have their pages marked UNSAFE at the first occurrence of one of its members in the enumeration. We can therefore be certain that a row, once concluded safe, will not be marked otherwise later. In addition, the enumeration property guarantees that if an edge row is found later in the pass, such as block 4 in Figure 6b, previous rows containing the same pages will be correctly "backmarked" as EDGE.

Pass 2: Gathering. ALIS now makes a second pass over the DRAM rows, searching for contiguous *row blocks* of unmarked pages, bordered on each side by rows with marked EDGE. We add each of these row blocks to a list while marking all their pages as USED. At the end of this step we have a complete list of all *guardable* memory areas immediately available for allocation. Note these row blocks are isolated from each other and all other system memory by a padding of at least one guard row.

Pass 3: Pruning. In this final cleanup pass, ALIS iterates through allocated pages, unmapping and returning to the OS pages that aren't marked as USED, freeing up any non-essential memory locked by the previous steps.

Reservation and Mapping. ALIS can now use the data structure obtained in the previous steps to allocate isolated buffers using one or more row blocks. Applications can map the (physical) pages in these buffers into the virtual address space at desired locations to satisfy the allocation request.

6.4 Implementation

We have implemented ALIS on top of Linux as a user-space library using 2518 lines of C code. ALIS reserves memory by mapping a file descriptor associated with anonymous shared memory (i.e., a `memfd` on Linux). ALIS uses the `/proc/self/pagemap` interface [35], to translate the buffer's virtual addresses to physical addresses. Finally, ALIS maps particular page frames into the process' virtual address space using the `mmap` system call by providing specific offsets into the `memfd` to specific virtual addresses using the `MAP_FIXED` flag. These mechanisms allow ALIS to seamlessly replace memory allocation routines used by applications with an isolated version. ALIS supports translation between the physical address space to the DRAM address space for the memory controller of all major CPU architectures.

7 Protecting Applications with ALIS

In this section, we show how we used ALIS to protect two popular applications that provide distributed key-value services, namely memcached [32] and HERD [34] against remote Rowhammer attacks. One key observation is that there are a few different ways to allocate space for the RDMA buffer. Since we are interested in isolating the memory used as the RDMA buffer for containing RDMA bit flips, it is crucial to understand the way each application manages memory for its RDMA buffers. Our allocator is capable of handling common cases such as when the memory is allocated with `mmap` or `posix_memalign` while it can be extended to support additional constructs. We now discuss the specifics of the applications which we tried with our custom allocator. We evaluate the performance of both systems when deployed using our custom allocator in §8.2.

7.1 Memcached

Many large-scale Internet services use DRAM-based key-value caches like memcached. Usually, memcached serves as a cache in front of a back-end database. Memcached with RDMA support [32] provides lower latency and higher throughput compared to the original memcached. This is done by introducing traditional RDMA-enabled set and get APIs. Given that memcached is a popular application, exploitable bit flips caused over the network as we discussed in §5 can affect many users.

RDMA-memcached is not open-source. We hence reverse engineered its binary to discover that it uses `posix_memalign` for allocating the RDMA buffers. Total size of these RDMA-buffers is approximately 5 MB. Unfortunately, instead of using the standard libc-provided `posix_memalign`, memcached-rdma uses its own statically-linked implementation. We hence needed to perform a simple binary instrumentation to instead jump to our implementation of `posix_memalign` which allocates isolated buffers.

7.2 HERD

HERD [34] is a key-value store that leverages RDMA to deliver low latency and high throughput. Unlike similar systems, HERD has been designed with RDMA in mind. The system offers clean RDMA primitives, and heavily relies on RDMA to reduce round-trip times, reduce latency, and maximize throughput. As a case-study, HERD is ideal, since simply turning RDMA off is not an option; the system is primarily designed around the concept of RDMA. Thus, if anyone needs to take advantage of the performance of HERD, they additionally need to secure its RDMA buffer, otherwise its users run the security risks of remote Rowhammer attacks.

HERD's initializer process uses `shmget` to allocate the RDMA buffer and share it with its worker process. While we could extend ALIS to support `shmget`, instead we opted to declare a global variable in HERD's initializer process to store allocated the RDMA buffer address and pass it to the worker process. This required modifying 10 lines of code in HERD.

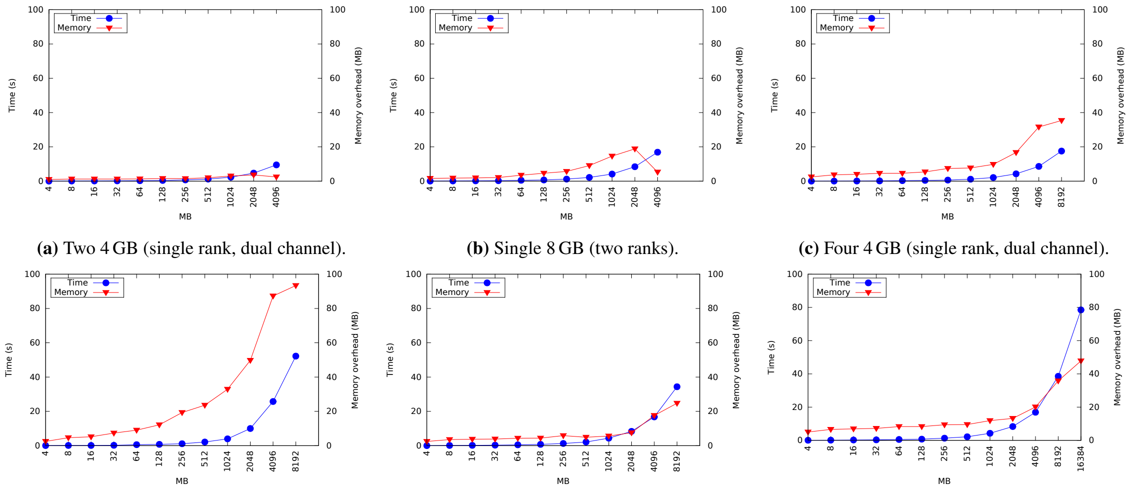
8 Evaluation

We use the same testbed that we used in §4 for our evaluation. Firstly we made sure ALIS was indeed protecting our system from bit flips. We modified Throwhammer's client to allocate isolated RDMA buffers using ALIS. Hammering remotely for extended periods of time with different strategies did not generate any bit flips outside the RDMA buffers unlike previously, thus enforcing the RDMA security model (§3).

We now evaluate in detail the memory overhead and performance impact of protecting applications.

8.1 Allocation Overhead

To calculate the overhead of ALIS, we wrote a simple test program that allocates an isolated buffer of a given size and reports how long it took for the allocation to succeed. Note that in most cases, we only pay this (modest) overhead once at application initialization time. Given that ALIS pools these allocations, in cases where an application re-allocates these buffers (e.g., [9, 46]), the subsequent allocations of the same size will be fast. We also collected statistics from our allocator on the number of extra pages that we had to allocate for guard rows.



(a) Two 4 GB (single rank, dual channel). **(b)** Single 8 GB (two ranks). **(c)** Four 4 GB (single rank, dual channel).

(d) Two 8 GB (two ranks, single channel). **(e)** Two 8 GB (two ranks, dual channel). **(f)** Four 8 GB (two ranks, dual channel).

Figure 7: The allocation time and memory overhead of isolating RDMA buffers of various sizes in different configurations.

Configurations We experimented with all possible configurations including multiple DRAM modules, channels, and ranks. We assume that up to half of the memory can be used for RDMA buffers, but nothing stops us from increasing this limit (e.g., 80%). We run each measurement 5 times and report the mean value.

Figure 7 shows two general expected trends in all configurations: 1. the allocation time increases as we request a larger allocation due to the required extra computation, 2. the amount of extra memory that our allocator needs for guard rows increases only modestly as we allocate larger safe buffers. In fact, the relative overhead becomes much smaller as we allocate larger buffers.

We also make a number of other observations:

1. The size of installed memory does not affect the allocation performance (Figure 7a vs. Figure 7c),
2. Increasing the number of ranks and channels increases the allocation time.
3. The number of ranks increases the allocation time more than the number of channels (Figure 7a vs. Figure 7b and Figure 7c vs. Figure 7d). Given that column address bits slice the DRAM address space into finer chunks than the channel bits, our allocator requires more computation to find safe memory pages when rank mirroring is active.

In general, allocating larger buffers slightly increases the amount of memory required for isolating the buffers given that our allocator stitches multiple safe *blocks* together to satisfy the requested size. Interestingly, as we allocate more memory, the allocator requires fewer areas and in some cases, this reduces the amount of memory required for guard rows (Figure 7a and Figure 7b).

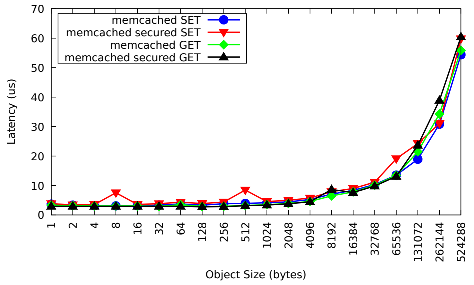


Figure 8: Secured memcached performance.

8.2 RDMA Performance

We now report on the performance implications of using ALIS on RDMA-memcached and HERD (§7). We use the same testbed that we used in our remote bit flip study (§4) and use the benchmarks provided by the applications. The benchmark included in RDMA-memcached measures the latency of SET and GET requests with varying value sizes. Figure 8 shows that our custom allocator only introduces negligible performance overhead in memcached-rdma. This is expected because ALIS only introduces a small overhead during initialization.

The benchmark included with HERD reports the throughput of HERD in terms of number of requests per second. Our measurements show that isolating RDMA buffers in HERD reduces the performance by 0.4% which is negligible. The original HERD paper [34] achieves the throughput of 26 million requests per second by using multiple client machines and a server machine with two processors. The authors of HERD verified that our throughput baseline is expected with our testbed. Hence, we conclude that ALIS does not incur any runtime overhead while isolating RDMA buffers.

9 Related work

Attacks Rowhammer was initially conceived in 2014 when researchers experimentally demonstrated flipping bits in DDR3 for x86 processors by just accessing other parts of memory [36]. Since then, researchers have proposed increasingly sophisticated Rowhammer exploitation techniques, on browsers [52, 15, 29], clouds [14, 51, 60], and ARM-based mobile platforms [59]. There have also been reports of bit flips on DDR4 modules [41, 59]. Finally, recent attacks have focused on bypassing state-of-the-art Rowhammer defenses [27, 56].

In all of these instances, the attacker needs to find a way to trigger the right bit flips that can alter critical data (page tables, cryptographic keys, etc.) and thus affect the security of the system. All these cases assume that the attacker has local code execution. In this paper, we showed how an adversary can induce bit flips by merely sending network packets.

Defenses Although we have plenty of advanced attacks exploiting bit flips, defenses are still behind. We stress here that for some of the aforementioned attacks that affect real products, vendors often disable software features. Linux kernel disabled unprivileged access to `pagemap` [35] in response to Seaborn's attack [52], Microsoft disabled deduplication [21] in response to the Dedup Est Machina attack [15], Google disabled the ION contiguous heap [58] in response to the Drammer attack [59] and further disabled high-precision GPU timers [4] in response to the GLitch attack [24]. A similar reaction to Throwhammer could be potentially disabling RDMA, which (a) in not realistic, and (b) does not solve the problem entirely. Therefore, we presented ALIS, a custom allocator that isolates a vulnerable RDMA buffer (and can in principle isolate any vulnerable to hammering buffer in memory). ALIS is quite practical, since, compared to other proposals [12, 16], it is completely implemented in user-space, compatible with existing software, and does not require special hardware features.

More precisely, CATT [16] can only protect kernel memory against Rowhammer attacks. We showed, however, that it is possible to target user applications with Rowhammer *over the network*. Furthermore, CATT requires kernel modification which introduce deployment issues (especially in the case of data centers). In particular, it applies a static partitioning between memory used by the kernel and the user-space. The kernel, however, often needs to move physical memory between different zones depending on the currently executing workload. In comparison, our proposed allocator is flexible, does not require modification to the kernel, and unlike CATT, can safely allocate memory by taking the physical to DRAM address space translation into account.

Another software-based solution, ANVIL [12] also lacks the translation information for implementing a proper protection. It relies on Intel's performance monitoring unit (PMU) that can capture precisely which physical addresses cause many cache misses. By accessing the neighboring rows of these physical addresses, ANVIL manually recharges victim rows to avoid bits to flip. An improved version of ANVIL with proper physical to DRAM translation can be an ideal software defense against remote Rowhammer attacks. Unfortunately, Intel's PMU (or AMD's) does not capture precise address information when memory accesses bypass the cache through DMA. Hence, our allocator can provide the necessary protection for remote DMA attacks (or even local DMA attacks [59]) while processor vendors extend the capabilities of their PMUs.

10 Conclusion

Thus far, Rowhammer has been commonly perceived as a dangerous hardware bug that allows attackers capable of executing code on a machine to escalate their privileges. In this paper, we have shown that Rowhammer is much more dangerous and also allows for remote attacks in practical settings. Remote Rowhammer attacks place different demands on both the attackers and the defenders. Specifically, attackers should look for new ways to massage memory in a remote system. Meanwhile, defenders can no longer prevent Rowhammer by banning local execution of untrusted code. We showed how an attacker can exploit remote bit flips in memcached to exemplify a remote Rowhammer attack. We further presented a novel defense mechanism that physically isolates the RDMA buffers from the rest of the system. This shows that while it may be hard to prevent Rowhammer bit flips altogether without wide-scale hardware upgrades, it is possible to contain their damage in software.

Disclosure

We have cooperated with the National Cyber Security Centre in the Netherlands to coordinate disclosure of the vulnerabilities to the relevant parties.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported in part by the MALPAY project and in part by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", and NWO 629.002.204 "Parallax".

References

[1] About H-series and compute-intensive A-series VMs for Windows. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/a8-a9-a10-a11-specs, Retrieved 31.05.2018.

[2] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types, Retrieved 31.05.2018.

[3] Compute Intensive Instances. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-hpc, Retrieved 31.05.2018.

[4] GLitch vulnerability status. http://www.chromium.org/chromium-os/glitch-vulnerability-status Retrieved 31.05.2018.

[5] Microsoft announces Windows 10 Pro for Workstations. https://blogs.windows.com/business/2017/08/10/microsoft-announces-windows-10-pro-workstations/, Retrieved 31.05.2018.

[6] Program for testing for the DRAM "rowhammer" problem. https://github.com/google/rowhammer-test, Retrieved 31.05.2018.

[7] Properties of a Physical-to-DRAM Address Mapping. https://www.vusec.net/download/?t=papers/dram-formal.pdf.

[8] Technical Info: A Deep Dive Into ProfitBricks. https://www.profitbricks.com/technical-info Retrieved 31.05.2018.

[9] MPI One-Sided Communication, 2014. https://software.intel.com/en-us/blogs/2014/08/06/one-sided-communication, Retrieved 31.05.2018.

[10] Advanced Micro Devices. *BIOS and Kernel Developers Guide (BKDG) for AMD Family 15h Models 60h-6Fh Processors*. May 2016.

[11] Andersen, S., and Abella, V. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[12] Aweke, Z. B., Yitbarek, S. F., Qiao, R., Das, R., Hicks, M., Oren, Y., and Austin, T. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. ASPLOS'16.

[13] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. OSDI'14.

[14] Bhattacharya, S., and Mukhopadhyay, D. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. CHESS'16.

[15] Bosman, E., Razavi, K., Bos, H., and Giuffrida, C. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. SP'16.

[16] Brasser, F., Davi, L., Gens, D., Liebchen, C., and Sadeghi, A.-R. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. SEC'17.

[17] Cai, Y., Ghose, S., Luo, Y., Mai, K., Mutlu, O., and Haratsch, E. F. Vulnerabilities in MLC NAND flash memory programming: experimental analysis, exploits, and mitigation techniques. HPCA'17.

[18] Cai, Y., Ghose, S., Luo, Y., Mai, K., Mutlu, O., and Haratsch, E. F. Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques. HPCA'17.

[19] Cock, D., Ge, Q., Murray, T., and Heiser, G. The Last Mile: An Empirical Study of Timing Channels on seL4. CCS'14.

[20] Costa, P., Ballani, H., Razavi, K., and Kash, I. R2C2: A Network Stack for Rack-scale Computers. SIGCOMM'15.

[21] CVE-2016-3272. Microsoft Security Bulletin MS16-092 - Important. *https: // technet. microsoft. com/ en-us/ library/ security/ ms16-092. aspx* (2016).

[22] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M. FaRM: Fast Remote Memory. NSDI'14.

[23] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. SOSP'15.

[24] Frigo, P., Giuffrida, C., Bos, H., and Razavi, K. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. SP'18.

[25] George, V., Piazza, T., and Jiang, H. Technology insight: Intel® next generation microarchitecture codename ivy bridge. In *Intel Developer Forum* (2011).

[26] Gras, B., Razavi, K., Bosman, E., Bos, H., and Giuffrida, C. ASLR on the Line: Practical Cache Attacks on the MMU. NDSS'17.

[27] Gruss, D., Lipp, M., Schwarz, M., Genkin, D., Juffinger, J., O'Connell, S., Schoechl, W., and Yarom, Y. Another flip in the wall of rowhammer defenses. In *S&P'18*.

[28] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. CCS'16.

[29] Gruss, D., Maurice, C., and Mangard, S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. DIMVA'16.

[30] Intel, I. Intel-64 and ia-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part 1*, 64 (2013).

[31] Jeong, E. Y., Woo, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., and Park, K. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. NSDI'14.

[32] Jose, J., Subramoni, H., Luo, M., Zhang, M., Huang, J., Wasi-ur Rahman, M., Islam, N. S., Ouyang, X., Wang, H., Sur, S., and Panda, D. K. Memcached Design on High Performance RDMA Capable Interconnects. ICPP'11.

[33] Kalia, A., Kaminsky, M., and Andersen, D. G. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. OSDI'16.

[34] Kalia, A., Kaminsky, M., and Andersen, D. G. Using RDMA Efficiently for Key-value Services. SIGCOMM'14.

[35] kernel, L. https://www.kernel.org/doc/Documentation/vm/pagemap.txt, Retrieved 31.05.2018.

[36] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. ISCA'14.

[37] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203* (2018).

[38] Kurmus, A., Ioannou, N., Papandreou, N., and Parnell, T. From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks. WOOT'17.

[39] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. Code-pointer Integrity. OSDI'14.

[40] Lanteigne, M. A Tale of Two Hammers: A Brief Rowhammer Analysis of AMD vs. Intel. http://www.thirdio.com/rowhammera1.pdf, May 2016.

[41] Lanteigne, M. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. http://www.thirdio.com/rowhammer.pdf, March 2016.

[42] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[43] Liu, F., Yin, L., and Blanas, S. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. EuroSys'17.

[44] Mitchell, C., Geng, Y., and Li, J. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. USENIX ATC'13.

[45] Mittal, R., Lam, V. T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. TIMELY: RTT-based Congestion Control for the Datacenter. SIGCOMM'15.

[46] Noronha, R., Chai, L., Talpey, T., and Panda, D. K. Designing NFS with RDMA for Security, Performance and Scalability. ICPP'07.

[47] Oliverio, M., Razavi, K., Bos, H., and Giuffrida, C. Secure Page Fusion with VUsion. SOSP'17.

[48] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. CCS'15.

[49] Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. SEC'16.

[50] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The Operating System is the Control Plane. OSDI'14.

[51] Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., and Bos, H. Flip Feng Shui: Hammering a Needle in the Software Stack. SEC'16.

[52] Seaborn, M., and Dullien, T. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. BHUS'15.

[53] Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Provost, J., Simmons, J., Tanda, E., Wanderer, J., Hölzle, U., Stuart, S., and Vahdat, A. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. SIGCOMM'15.

[54] STANDARD, J. DDR3 SDRAM. JESD79-3C, Nov 2008.

[55] Tang, J., and Team, T. M. T. S. Exploring control flow guard in windows 10. *http://blog.trendmicro.com/trendlabs-security-intelligence/exploring-control-flow-guard-in-windows-10, Retrieved 31.05.2018.*

[56] Tatar, A., Razavi, K., Bos, H., and Giuffrida, C. Defeating software mitigations against rowhammer: a surgical precision hammer. In *RAID'18.*

[57] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, U., Lozano, L., and Pike, G. Enforcing Forward-edge Control-flow Integrity in GCC and LLVM. SEC'14.

[58] Tjin, P. android-7.1.0_r7 (Disable ION_HEAP_TYPE_SYSTEM_CONTIG). *https://android.googlesource.com/device/google/marlin-kernel/+/android-7.1.0_r7* (2016).

[59] van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., and Giuffrida, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. CCS'16.

[60] Xiao, Y., Zhang, X., Zhang, Y., and Teodorescu, R. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. SEC'16.

[61] Yarom, Y., and Falkner, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. SEC'14.

[62] Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M. H., and Zhang, M. Congestion Control for Large-Scale RDMA Deployments. SIGCOMM'15.

Varys

Protecting SGX Enclaves From Practical Side-Channel Attacks

Oleksii Oleksenko[†], Bohdan Trach[†], Robert Krahn[†], Andre Martin[†],
Christof Fetzer[†], Mark Silberstein[‡]
[†]*TU Dresden,* [‡]*Technion*

Abstract

Numerous recent works have experimentally shown that Intel Software Guard Extensions (SGX) are vulnerable to cache timing and page table side-channel attacks which could be used to circumvent the data confidentiality guarantees provided by SGX. Existing mechanisms that protect against these attacks either incur high execution costs, are ineffective against certain attack variants, or require significant code modifications.

We present **Varys**, a system that protects unmodified programs running in SGX enclaves from cache timing and page table side-channel attacks. Varys takes a pragmatic approach of strict reservation of physical cores to security-sensitive threads, thereby preventing the attacker from accessing shared CPU resources during enclave execution. The key challenge that we are addressing is that of maintaining the core reservation in the presence of an untrusted OS.

Varys fully protects against all L1/L2 cache timing attacks and significantly raises the bar for page table side-channel attacks—all with only 15% overhead on average for Phoenix and PARSEC benchmarks. Additionally, we propose a set of minor hardware extensions that hold the potential to extend Varys' security guarantees to L3 cache and further improve its performance.

1 Introduction

Intel Software Guard Extensions (SGX) enclaves provide a shielded environment to securely execute sensitive programs on commodity CPUs in the presence of a privileged adversary. So far, no successful direct attack on SGX has been reported, i.e., none that compromises SGX's security guarantees. However, numerous works demonstrate that SGX is vulnerable to several types of side channel attacks (SCAs), in particular, traditional cache timing and page table SCA that reveal page-level memory accesses [9, 39, 21, 47, 43, 45, 24], as well as speculative attacks [29, 12] that use the side channels as a way of retrieving information. Although Intel explicitly excludes side channels from the SGX threat model, SCAs effectively circumvent the SGX confidentiality guarantees and impede SGX adoption in many real-world scenarios. More crucially, a privileged adversary against SGX can mount much more powerful SCAs compared to the unprivileged one in canonical variants of the attacks. For example, a malicious OS can dramatically reduce the noise levels in cache timing attacks via single-stepping [24] or by slowing down the victim.

In this paper, we investigate *practical* ways of protecting SGX programs from page table and cache timing SCAs. Specifically, we focus on the case where unmodified general-purpose applications are executed in enclaves in order to protect their secrets, as it is the case with Haven [4], Graphene-SGX [11], or SCONE [3].

We postulate that a *practical* solution should have low performance overheads, require no modifications to application source code, and impose no restrictions on the application's functionality (such as restricting multithreading). We assume that recompilation of source code is acceptable as long as it does not require code changes.

Existing mitigation techniques, however, fall short of satisfying our requirements. Traditional hardening techniques against cache timing attacks [5, 23] require rewriting the application; recent defenses [22] based on Intel TSX technology also require code changes; memory accesses obfuscation approaches, such as DR.SGX [8], incur high performance cost (at least 3× and up to 20×); T-SGX [40] prevents controlled OS attacks, but it is ineffective against concurrent attacks on page tables and caches. Déjà Vu [14] protects only against page table attacks and is prone to false positives.

In *Varys*, we strive to achieve both application performance and user convenience while making page table and cache timing SCAs on enclaves much harder or entirely impossible to mount. The basic idea behind Varys design is *trust but verify*. A potentially malicious OS is *requested* to execute the enclave in a *protected environment* that prevents all known cache timing and page fault attacks on

SGX enclaves. However, the Varys trusted runtime inside the enclave verifies that the OS fulfills the request.

Our main observation is that all published page table and L1/L2 cache timing attacks on SGX require either (1) a high rate of enclave exits, or (2) control of a sibling hyperthread on the same CPU core with the victim. Consequently, if an enclave is guarded against frequent asynchronous exits and executes on a dedicated CPU core without sharing it with untrusted threads, it would be protected against the attacks. The primary challenge that Varys addresses is in maintaining such a protected environment in face of a potentially malicious OS. It achieves this goal via two mechanisms: *asynchronous enclave exits monitoring* and *trusted reservation*.

First, Varys monitors when asynchronous enclave exits (AEX) occur (e.g., for scheduling another process on the core or handling an exception) and restricts the frequency of such exits, terminating the enclave once the AEX frequency bound is exceeded. Varys sets the bound to the values that render all known attacks impossible. Notably, the bound is much higher than the frequency of exits in an attack-free execution, thereby minimizing the chances of false positives as we explain in §4.

Second, Varys includes a mechanism for *trusted core reservation* such that the attacker cannot access the core resources shared with the enclave threads while they are running, nor can it recover any secrets from the core's L1 and L2 caches afterward. For example, consider an in-enclave execution of a multi-threaded application with two threads. Assuming a standard processor with hyperthreading (SMT), all it takes to prevent concurrent attacks on L1/L2 caches is to guarantee that the two enclave threads always run *together* on the same physical core. As a result, the threads occupy both hardware threads of the core, and the attacker cannot access the core's caches. Note that this simple idea prevents any concurrent attacks on core's resources shared between its hyperthreads, such as branch predictor and floating point unit. It also prevents exit-less SCAs on page table attributes [43] because they require attacker's access to the core's TLB—available only if the attacker thread is running on that core. Additionally, to ensure that the victim leaves no traces in the caches when de-scheduled from the core, Varys explicitly evicts the caches when enclave threads are preempted.

While conceptually simple, the implementation of the trusted reservation mechanism is a significant challenge. An untrusted OS may ignore the request to pin two enclave threads to the same physical core, may re-enable CPU hyperthreading if disabled prior to enclave execution, and may preempt each of the enclave threads separately in an attempt to break Varys's defense.

Our design offers *a low-overhead mechanism for trusted core reservation under an untrusted OS*. Application threads are grouped in pairs, and the OS is *requested* to co-locate these pairs on the same physical CPU core. The trusted application threads are instrumented (via a compiler pass) to *periodically verify* that they are indeed co-scheduled and running together on the same core. Varys terminates the enclave if co-scheduling is violated or if any of the threads in the pair gets preempted too often. To reduce the frequency of legitimate exits and lower the false positives, Varys uses exitless system calls [3] and in-enclave thread scheduling such that multiple application threads can share the same OS thread. Moreover, Varys configures the OS to reduce the frequency of interrupts routed to the core in order to avoid interference with attack-free program execution. However, if the OS ignores the request, this will effectively lead to denial of service without compromising the enclave's security.

Varys primarily aims to protect multi-threaded programs by reserving complete cores and scheduling the application threads on them, i.e., protection against SCAs translates into an allocation policy that allocates or frees computing resource with a granularity of one core. We believe that Varys exercises a reasonable trade-off between security and throughput for services that require the computational power of one or more cores. For single-threaded applications Varys pairs the application thread with a service thread to reserve the complete core.

Due to the lack of appropriate hardware support in today's SGX hardware, Varys remains vulnerable to timing attacks on Last Level Cache (LLC) as we explain in §8. We suggest a few minor hardware modifications that hold the potential to solve this limitation and additionally, eliminate most of the runtime overhead. These extensions allow the operating system to stay in control of resource allocations but permit an enclave to determine if its resource allocation has changed.

Our contributions include:

- Analysis of attack requirements.
- A set of measures that can be taken to protect against these attacks using existing OS features.
- Varys, an approach to verifying that the OS correctly serves our request for a protected environment.
- Implementation of Varys with 15% overhead across PARSEC [7] and Phoenix [38] benchmark suites.
- Proposal for hardware extensions that improve Varys's security guarantees and performance.

2 Background

2.1 Intel SGX

Intel Software Guard Extensions is an ISA extension that adds hardware support for Trusted Execution Environments. SGX offers creation of *enclaves*—isolated memory regions, with code running in a novel enclave execution mode. Also, Intel SGX provides a local and remote attestation systems that is used to establish whether the software is running on an SGX-capable CPU.

SGX has hardware-protected memory called Enclave Page Cache (EPC). Accesses to the EPC go through the Memory Encryption Engine (MEE), which transparently encrypts and applies MAC to cache lines on writes, and decrypts and verifies cache lines on reads. Access permissions of the pages inside an enclave are specified both in page tables and in EPC Metadata, and permissions can be only restricted via page tables. The enclave's memory contents in the caches are stored as plaintext.

Enclaves can use more virtual memory than can be stored by the EPC. In this case, EPC paging will happen when a page not backed by physical memory is accessed. The CPU, in coordination with the OS kernel, can evict a page to untrusted memory. Currently, the EPC size available to user applications is around 94 MB.

2.2 Side-channel attacks

In this work, we focus mainly on cache timing and page table attacks as they provide the widest channel and thus, are the most practical to be used to attack enclaves.

Cache timing attacks [36, 32, 27, 25, 2, 48] infer the memory contents or a control flow of a program by learning which memory locations are accessed at fine granularity. The adversary uses the changes in the state of a shared cache as a source of information. In particular, she sets the cache to a predefined state, lets the victim interact with the cache for some time, and then reads from the cache again to determine which parts were used by the victim. Accordingly, for this attack to work, the adversary has to be able to access the victim's cache.

Page table attacks reveal page-level access patterns of a program. They are usually considered in the context of trusted execution environments as they are possible only if privileged software is compromised, hence they are also called *controlled-channel attacks* [47].

These attacks can be classified into *page-fault based* and *page-bit based*. Page-fault based attacks [47, 41] intercept all page-level accesses in the enclave by evicting the physical pages from the EPC. Page-bit based attacks [45, 43] use the *Accessed* and *Dirty* page table bits as an indication of access to the page, without page faults. However, these bits are cached in a TLB, so to achieve the required fidelity, the adversary has to do both, i.e., to clear the flags and to flush the victim's TLB.

3 Threat Model

We assume the standard SGX threat model. The adversary is in complete control of privileged software, in particular, the hypervisor and the operating system. She can spawn and stop processes at any point, set their affinity and modify it at runtime, arbitrarily manipulate the interrupt frequency and cause page faults. The adversary can also read and write to any memory region except the enclave memory, map any virtual page to any physical one and dynamically re-map pages. Together, it creates lab-like conditions for an attack: it could be running in a virtually noise-free environment.

4 System footprint of SGX SCAs

In this section we analyze the runtime conditions required for the known SCAs to be successful. Varys mitigates the SCAs by executing an enclave in a protected environment and preventing these conditions from occurring.

Cache attacks [36, 42, 32, 2, 48, 50, 10, 17] can be classified into either *concurrent*, i.e, running in parallel with the victim, or *time-sliced*, i.e., time-sharing the core (or a hyperthread) with the victim.

Time-sliced cache attacks $\Longrightarrow$ high AEX rate. For a time-sliced attack to be successful, the runtime of the victim in each time slice must be short; otherwise, the cache noise will become too high to derive any meaningful information. For example, the attack described by Zhang et al. [49] can be prevented by enforcing minimal runtime of 100us [44], which translates into 10kHz interrupt rate. It is dramatically higher than the preemption rate under normal conditions—below 100Hz (see §5.3). If the victim is an enclave thread, its preemption implies an asynchronous enclave exit (AEX).

Concurrent cache attacks $\Longrightarrow$ shared core. The adversary running in parallel with the victim must be able to access the cache level shared with it. Thus, L1/L2 cache attacks are not possible unless the adversary controls a sibling hyperthread.

We note that with the availability of the Cache Allocation Technology (CAT) [26], the share of the Last Level Cache (LLC) can also be allocated to a hardware thread, preventing any kind of concurrent LLC attacks [31]. However, this defense is ineffective for SGX because the allocation is controlled by an untrusted OS. We suggest one possible solution to this problem in §8.

Page-fault page table attacks $\Longrightarrow$ high AEX rate. These attacks inherently increase the page fault rate, and consequently AEX rate, as they induce page faults to infer the accessed addresses. For example, as reported by Wang et al. [45], a page table attack on EdDSA requires approximately 11000 exits per second. In fact, high exit rates have been already used as an attack indicator [40, 22].

Interrupt-driven page-bit attacks $\Longrightarrow$ high AEX rate. If the attacker does not share a physical core with the victim, these attacks incur a high exit rate because the attacker must flush the TLB on a remote core via Inter-Processor Interrupts (IPIs). The rate is cited to be around 5500Hz [43, 45]. While lower than other attacks, it is still above 100Hz experienced in attack-free execution (§5.3).

Exit-less page-bit attacks $\Longrightarrow$ shared core. The only way to force TLB flushes without IPIs is by running an adversary sibling hyperthread on the same physical core to force evictions from the shared TLB [45]. These attacks involve no enclave exits, thus are called *silent*.

In summary, all the known page table and L1/L2 cache timing SCAs on SGX rely on (*i*) an abnormally high rate of asynchronous enclave exit, or/and (*ii*) an adversary-controlled sibling hyperthread on the same physical core. The only exception is the case when the victim has a slowly-changing working set, which we discuss in §7.2. These observations drive the design of the Varys system we present next.

5 Design

Varys provides a *side-channel protected execution environment* for SGX enclaves. This execution environment ensures that neither time-sliced nor concurrent cache timing as well as page table attacks can succeed. To establish such an environment, we (*i*) introduce a trusted reservation mechanism, (*ii*) combine it with a mechanism for monitoring enclave exits, and (*iii*) present a set of techniques for reducing the exit rate in an attack-free execution to avoid false positives.

5.1 Trusted reservation

The simplest way to ensure that an adversarial hyperthread cannot perform concurrent attacks on the same physical core would be to disable hyperthreading [33]. However, doing so not only hampers application performance but may not be reliably verified by the enclave: One can neither trust the operating system information nor can one execute the CPUID instruction inside of enclaves.

An alternative approach is to allow sharing of a core only by benign threads. Considering that in our threat model only the enclave is trusted, we allow core sharing only among the threads from the same enclave. We can achieve this goal by dividing the application threads in pairs (if the number of threads is odd, we spawn a dummy thread) and requesting the OS to schedule the pairs on the same cores. Since we do not trust the OS, we ensure collocation by establishing a covert channel in the L1 cache as follows.

The idea is to determine whether the threads share L1 cache or only last level cache. The later would imply the threads are on different physical cores. We refer to the procedure that determines the threads co-location on the core as *handshake*. To perform the handshake, we establish a simple covert channel between the threads via L1: One of the two threads writes a dummy value to a shared memory location, thus, forcing it to L1. Then, the sibling thread reads the same memory location and measures the timing. If the reading is fast (up to 10 cycles per read), both threads use the same L1 cache, otherwise (more than 40 cycles) they share only LLC, implying they are on different cores. If the threads indeed run on different cores, the OS did not satisfy the scheduling request made by Varys. We conclude that the enclave is under attack and terminate it. Since the current version of SGX does not provide trusted fine-grain time source, we implement our own as we explain in §6.2.

Of course, immediately after we established that the two threads are executing on the same core, the operating system could reschedule these threads on different cores. However, this rescheduling would cause an asynchronous enclave exit (AEX), which we detect via AEX monitoring as we discuss next.

5.2 AEX monitoring

To detect an asynchronous enclave exit, we monitor the SGX State Save Area (SSA). The SSA is used to store the execution state of an enclave upon an AEX. Since some parts of the enclave execution state are deterministic, we can detect an AEX by overwriting one of the SSA fields with an invalid value and monitoring it for changes.

For example, in the current implementation of SGX, the EXIT_TYPE field of an SSA frame is restricted to values 011b and 110b [26]. Thus, if we write 000b to EXIT_TYPE, SGX will overwrite it with another, predefined value at the next AEX. To detect an AEX, we periodically read and compare this field with the predefined value. Note that it is not the only SSA field that we could use for this purpose; many other registers, such as Program Counter, could be used too.

Now that we have a detection mechanism, it is sufficient to count the AEX events and abort the application if they are too frequent. Yet, to calculate the frequency, we need a trusted time source which is not available inside an enclave. Fortunately, precise timing is not necessary for this particular purpose as we would only use the time to estimate the number of instructions executed between AEXs. It is possible to estimate it through the AEX monitoring routine that our compiler pass adds to the application. Since it adds the routine every few hundred LLVM IR instructions, counting the number of times it is called serves a natural counter of LLVM IR instructions. In Varys, we define the AEX rate as number of AEXs per number of executed IR instructions.

Even though IR instructions do not correspond to machine instructions, one IR instruction maps on average to less than one x86-64 machine instruction[1]. Thus, we overestimate the AEX rate, which is safe from the security perspective.

Originally, we considered using TSX (Transactional Synchronization Extensions) to detect AEXs—similar to the approach proposed by Gruss et al. [22]. The main limitation of TSX is, however, that it does not permit

[1] In our experience with Phoenix and PARSEC benchmark suites, calling the monitoring routine every 100 IR instructions resulted in the polling period of 70–150 cycles.

non-transactional memory accesses within transactions. Hence, a) handshaking is not possible within a TSX transaction—this would lead to a transaction abort, and b) the maximum transaction length is limited and we would need to split executions in multiple transactions.

5.3 Restricting Enclave Exit Frequency

Ensuring that protected applications exit as rarely as possible is imperative for our approach. If the application has a high exit rate under normal conditions, not only does it increase the overhead of the protection, but also makes it harder to distinguish an attack from the attack-free execution. In the worst case, if the application's normal exit rate is sufficiently high (i.e., more than 5500 exits/second, see below), the adversary does not have to introduce any additional exits and can abuse the existing ones to retrieve information. Therefore, we have to analyze the sources of exits and the ways of eliminating or reducing them.

Under SGX, an application may exit the enclave for one of the following reasons: when the application needs to invoke a system call; to handle system timer interrupts, with up to 1000 AEX/s, depending on the kernel configuration; to handle other interrupts, which could happen especially frequently if Varys runs with a noisy neighbor (e.g., a web server); to perform EPC paging when the memory footprint exceeds the EPC size; to handle minor page faults, which could happen frequently if the application works with large files.

We strive to reduce the number of exits as follows. We use asynchronous exit-less system calls implemented, for example, in Eleos [35] and SCONE [3] (which we use in our implementation). Further, we combine asynchronous system calls with user-level thread scheduling inside the enclave to avoid reliance on the OS scheduling. We avoid the timer interrupt by setting the timer frequency to the lowest available —100 Hz—and enabling the DynTicks feature. Regular interrupts are re-routed to non-enclave cores. Last, we prevent minor page faults when accessing untrusted memory via `MAP_POPULATE` flag to `mmap` calls.

To evaluate the overall impact of these changes, we measure the exit frequencies of the applications used in our evaluation (see §7 for the benchmarks' description). The results are shown in Figure 1.

As we see, the rate is (i) relatively stable across the benchmarks and (ii) much lower than the potential attack rate of more than 1000 exits per second. Specifically, the attack presented by Van Bulck et al. [43] has one of the lowest interrupt rates among the published time-sliced attacks. We ran the open-sourced version of the attack and observed the rate of at least 5500 exits per second, which is in line with the rate presented in the paper. Correspondingly, if we detect (see §6.2) that the AEX rate is getting above 100 Hz, we can consider it a potential attack and take appropriate measures. To avoid false positives, we could set the threshold even higher—around 2kHz—without compromising security (see §7.2).

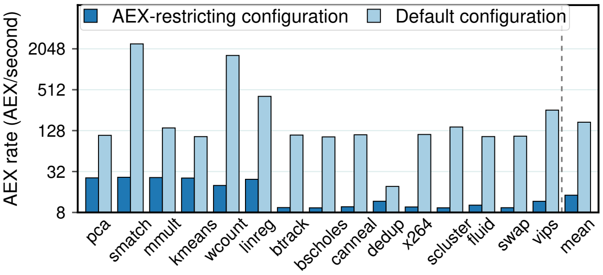


Figure 1: AEX rates under normal system configuration and with re-configured system.

```
while(true):
  wait_for_request()
  if (secret == 0): response = ∗a
  else: response = ∗b
```

Figure 2: An example of code leaking information in cache side-channel even with low frequency of enclave exits. If a and b are on different cache lines and the requests are coming infrequently, it is sufficient to probe the cache at the default frequency of OS timer interrupts.

5.4 Removing residual cache leakage

As we explained in §4, even with low frequency of enclave exits some leakage will persist if the victim has a slowly changing working set. Consider the example in Figure 2: the replies to user requests depend on the value of a secret. If requests arrive infrequently (e.g., 1 per second), restricting the exit frequency would not be sufficient; even if we set the bar as low as 10 exits per second (the rate we achieved in §5.3), the victim will touch only one cache line and thus, will reveal the secret.

To completely remove the leakage at AEX, we should flush the cache before we exit the enclave. This would remove any residual cache traces that an adversary could use to learn whether the enclave has accessed certain cache lines. Unfortunately, this operation is not available at user-space on Intel CPUs [26] nor do we have the possibility to request a cache flush at each AEX. Moreover, Ge et al. [18] have proven that the kernel-space flush commands do not flush the caches completely. `CLFUSH` instruction does not help either as it flushes a memory address, not the whole cache set. Thus, it cannot flush the adversary's eviction set residing in a different virtual address space, as it is the case in Prime+Probe attacks.

Instead, on each enclave entry, we write a dummy value to all cache lines in a continuous cache-sized memory region (e.g., 32KB for L1), further called *eviction region*. In case of L1, for which instruction and data are disjoint, we also execute a 32KB dummy piece of code to evict the instruction cache. This way, regardless of what the victim

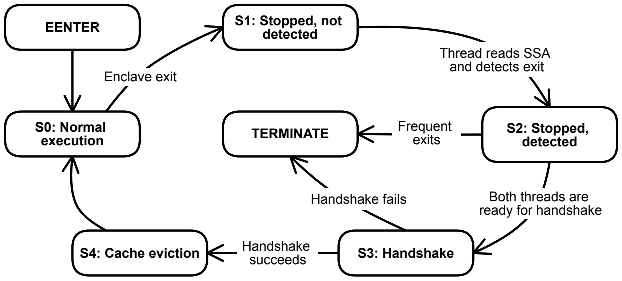


Figure 3: State diagram of a Varys-protected application.

does in between the exits, external observer will see that all the cache sets and all the cache ways were accessed and no information will be leaked.

6 Implementation

We implement Varys as an LLVM compiler pass that inserts periodic calls to a runtime library. We use SCONE to provide us with asynchronous system calls as well as in-enclave threading such that we minimize the need for an application to exit the enclave.

6.1 LLVM compiler pass

The cornerstone of Varys is the enclave exit detection. As discussed in §5.2, it requires all application threads to periodically poll the SSA region. Although we implement the checks as a part of a runtime library (§6.2), calls to the library have to be inserted directly into the application. To do this, we instrument the application using LLVM [30].

The goal of the instrumentation pass is to call the library and do the SSA polling with a predictable and configurable frequency. We achieve it by inserting the following sequence before every basic block: We increment a counter by the length of the basic block (in LLVM IR instructions), call the library if the counter reached a threshold, and skip the call otherwise. If the basic block is longer than the threshold, we add several calls. This way, the checks will be performed each time the application executes a given (configurable) number of IR instructions. We also reserve one of the CPU registers for the counter, as it is manipulated every few instructions and having the counter in memory would cause much higher overheads.

A drawback of SSA polling is that it has a blind zone. If a malicious OS preempts a thread multiple times in a very short period of time, they may happen before the counter reaches the threshold and the thread checks the SSA value. Hence, they will be all counted as a single enclave exit. This allows an adversary to launch stealthy cache attacks on small pieces of code by issuing occasional series of frequent preemptions. Yet, this vulnerability would be hard to exploit because the blind zone is narrow—on the order of dozens of cycles, depending on the configuration—and the adversary must run in tight synchronization with the victim to retrieve any meaningful information.

Optimization. Adding even a small piece of code to every basic block could be expensive as the blocks themselves are often only 4–5 instructions long. We try to avoid this by applying the following optimization.

Consider a basic block B0 with two successors, B1 and B2. In a naive version, in the beginning of each basic block we increment the IR instruction counter by the length of the corresponding basic block. However, if B0 cannot jump into itself, it will always proceed to a successor. Therefore, it is sufficient to increment the counters only in the beginnings of B1 and B2 by, accordingly, `length(B0)+length(B1)` and `length(B0)+length(B2)`. If B1 or B2 have more than one predecessor, it could lead to overestimation and more frequent SSA polling, which only reduces the blind zone.

6.2 Runtime library

Most of Varys' functionality is contained in a runtime library implementing the state machine in Figure 3.

When a program starts, it begins *normal execution* (S0). As long as the program is in this state, it counts executed instructions thus simulating a timer.

When one of the threads is interrupted, the CPU executes an AEX and overwrites the corresponding SSA (S1). As its sibling thread periodically polls the SSA, it eventually detects the exit. Then, if the program has managed to make sufficient progress since the last AEX (i.e., if the IR instruction counter has a large enough value), it transfers to the *detected* state (S2). Otherwise, the program terminates. To avoid false positives, we could terminate the program only if it happens several times in a row.

In S2, the sibling declares that the handshake is pending and starts busy-waiting. When the first thread resumes, it detects the pending handshake, and the pair enters state S3. If the handshake fails, the program is terminated[1]. Otherwise, one of the threads evicts L1 and L2 caches, and the pair continues normal execution.

Software timer. To perform cache measurements during the handshake phase, we need a trusted fine-grained source of time. Since the hardware time counter is not available in the current version of SGX, we implement it in software (similar to Schwarz et al. [39]). We spawn an enclave thread incrementing a global variable in a tight loop, giving us approximately one tick per cycle.

However, the frequency of the software timer is not reliable. An adversary can abuse the power management features of modern Intel CPUs and reduce the timer tick frequency by reducing the operational frequency of the underlying core. If the timer becomes slow enough, the handshake will be always succeeding. To protect against

[1] In practice, timing measurements are noisy and the handshake may fail for benign reasons. Therefore, we retry it several times and consider it failed only if the timing is persistently high.

```
.align 64
label1: jump label2 // jump to the next cache line
.align 64
label2: jump label3
```

Figure 4: A code snippet evicting cache lines in the L1 instruction cache. For evicting a 32 KB cache, the pattern is repeated 512 times.

it, we measure the timing of a constant-time operation (e.g., a series of in-register additions). Then, we execute the handshake only if the measurement matches the expected value.

Instruction cache eviction. Writing to a large memory region is not sufficient for evicting L1 or L2 caches. L1 has distinct caches for data (L1d) and instructions (L1i), and L2 is non-inclusive, which means that evicting L2 does not imply evicting L1i. Hence, the attacks targeting execution path are still possible.

To evict L1i, we have to execute a large piece of code. The fastest way of doing so is depicted in Figure 4. The code goes over a 32 KB region and executes a jump for each cache line thus forcing it into L1i.

L2 cache eviction. Evicting L2 cache is not as straightforward as L1 as it is physically-indexed physically-tagged (PIPT) [46]. For the L2 cache, allocating and iterating over a continuous virtual memory region does not imply access to continuous physical memory, and therefore does not guarantee cache eviction. A malicious OS could apply cache colouring [6, 28] to allocate physical pages in a way that the vulnerable memory locations map to one part of the cache and the rest of the address space—to another. This way, the vulnerable cache sets would not be evicted, and the leakage would persist.

With L2 cache, we do two passes over the eviction region. The first time, we read the region to evict the L2 cache. The second time, we read and measure the timing of this read. If the region is continuous, the first read completely fills the cache and the second read should be relatively fast as all the data is in the cache. However, if it is not the case, some pages of the eviction region would be competing for cache lines and evicting each other, thus making the second read slower. We use this as an indicator that L2 eviction is not reliable and we should try to allocate another region. If the OS keeps serving us non-continuous memory, we terminate the application as the execution cannot be considered reliable anymore.

6.3 SCONE

We base our implementation on SCONE [3], a shielding framework for running unmodified application inside SGX enclaves. Among other benefits, SCONE provides two features that make our implementation more efficient and compact. First, it implements user-level threading, which significantly simplifies thread pairing. As the number of enclave threads is independent of the number of application threads and fixed, it suffices to allocate and initialize thread pairs at program startup. Second, it provides asynchronous system calls. They not only significantly reduce the rate of enclave exits but also make this rate more predictable and application agnostic.

We should note, that Varys is not conceptually linked to SCONE. We could have avoided user-level threading by modifying the standard library to dynamically assign thread pairs. The synchronous system calls are also not an obstacle, but they require a mechanism to distinguish different kinds of enclave exits.

7 Evaluation

In this section, we measure the performance impact of Varys, the efficiency of attack detection and prevention, as well as the rate of false positives.

Applications. We base our evaluation on the Fex [34] evaluation framework, with PARSEC [7] and Phoenix [38] benchmark suites as workloads. The following benchmarks were excluded: *raytrace* depends on the dynamic X Window System libraries not shipped together with the benchmark; *freqmine* is based on OpenMP; *facesim* and *ferret* fail to compile under SCONE due to position-independent code issues. Together with the benchmarks, we recompile and instrument all the libraries they depend upon. We also manually instrument the most frequently used *libc* functions so that at least 90% of the execution time is spend in a protected code. We used the largest inputs that do not cause intensive EPC paging as otherwise, they could lead to frequent false positives.

Methodology. All overheads were calculated over the native SGX versions build with SCONE. The reported results are averaged over 10 runs and the "mean" value is a geomean across all the benchmarks.

Testbed. We ran all the experiments on a 4-core (8 hyperthreads) Intel Xeon CPU operating at 3.6 GHz (Skylake microarchitecture) with 32 KB L1 and 256 KB L2 private caches, an 8 MB L3 shared cache, 64 GB of RAM, and a 1TB SATA-based SSD. The machine was running Linux kernel 4.14. To reduce the rate of enclave exits, we configure the system as discussed in §5.3.

7.1 Performance Evaluation

Runtime. Figure 5 presents runtime overheads of different Varys security features. On average, the overhead is ~15%, but it varies significantly among benchmarks.

A major part of the overhead comes from the AEX detection, which we implement as a compiler pass. Since the instrumentation adds instructions that are not data dependent on the application's data flow, they can run in parallel. Therefore, they highly benefit from instruction

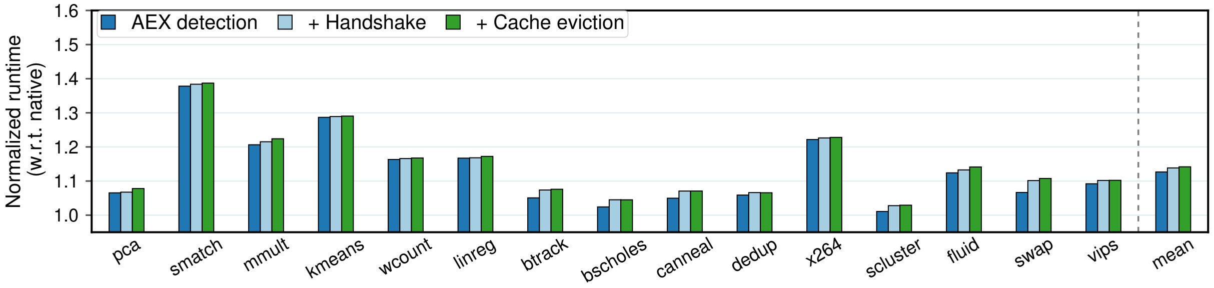


Figure 5: Performance impact of Varys security features with respect to native SGX version. Each next bar includes all the previous features. (Lower is better.)

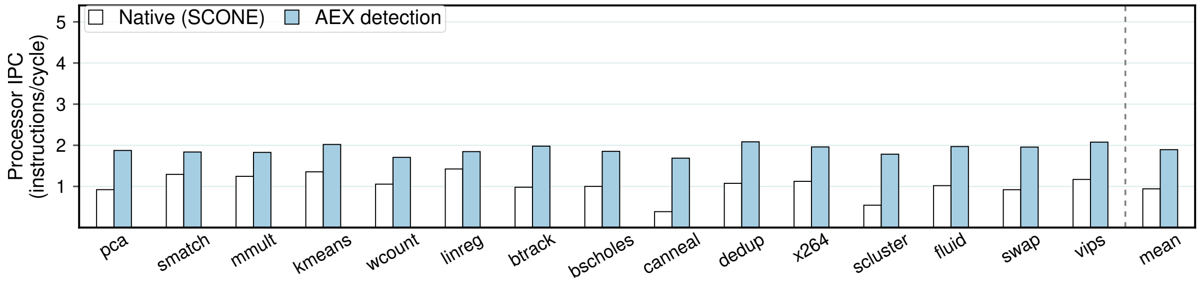


Figure 6: IPC (instructions/cycle) numbers for native and protected versions.

level parallelism (ILP), which we illustrate with Figure 6. The applications that have lower ILP utilization in the native version (e.g., *canneal* and *stream cluster*) can run a larger part of the instrumentation in parallel, thus amortizing the overhead.

Since we apply instrumentation per basic block, another factor that influences the overhead is the average size of basic blocks. The applications dominated by long sequences of arithmetic operations (e.g., *linear regression*) tend to have longer basic blocks and lower number of additional instructions (53% in this case), hence the lower overhead. At the same time, the applications with tight loops on the hot path cause higher overhead. Therefore, *string match* has higher overhead than *kmeans*, even though they have approximately the same level of IPC.

The second source of overhead is trusted reservation. It does not cause a significant slowdown because the handshake protocol is relatively small, including ten memory accesses for the covert channel and the surrounding code for the measurement. The overhead could be higher as the headshake is synchronized, i.e., two threads in a pair can make progress only if both are running. Otherwise, if one thread is descheduled, the second one has to stop and wait. Yet, as we see in Figure 5, it happens infrequently.

Finally, cache eviction involves writing to a 256 KB data region and executing a 32 KB code block. Due to the pseudo-LRU eviction policy of Intel caches, we have to repeat the writing several times (three, in our case). Together, it takes dozens of microseconds to execute, depending on the number of cache misses. Fortunately, we evict only after enclave exits, which are infrequent under normal conditions (§5.3) and the overhead is low.

Multithreading. As Varys is primarily targeted at multithreaded applications, it is crucial to understand its impact on multithreaded performance. To evaluate this parameter, we measured the execution time of all benchmarks with 2, 4, and 8 threads with respect to native versions with the same number of threads. Mind that these are user-level threads; the number of underlying enclave threads is always 4. The results are presented in Figure 7.

Generally, Varys does not have a significant impact on multithreaded scaling. However, there are a few exceptions. First, larger memory consumption required for multithreading causes EPC paging, thus increasing the AEX rate and sometimes even causing false positives. We can see this effect in *dedup* and *x264*: the higher AEX rate makes the flushing more expensive and eventually leads to false positives with higher numbers of threads. For the same reason, we excluded *linear regression*, *string match*, and *word count* from the experiment.

Another interesting effect happens in multithreaded *kmeans*. The implementation of *kmeans* that we use frequently creates and joins threads. Internally, `pthread_join` invokes memory unmapping, which in turn causes a TLB flush and an enclave exit. Correspondingly, the more threads *kmeans* uses, the more AEXs appear and the higher is the overhead.

Case Study: Nginx. To illustrate the impact of Varys on a real-world application, we measured throughput and latency of Nginx v1.13.9 [1] using ab benchmark. Nginx was running on the same machine as previous experiments

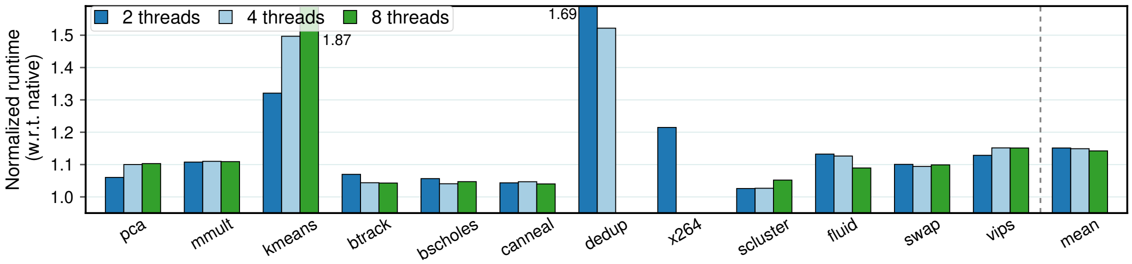


Figure 7: Runtime overhead with different number of threads. (Lower is better.)

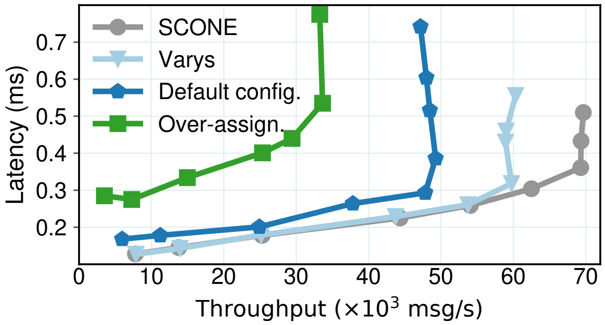


Figure 8: Throughput-latency plots of Nginx. Varys: low-exit system configuration, Default conf.: default configuration of Linux, Over-assign.: another process is competing for a core with Nginx.

Time threshold, SW timer ticks	**False positives, %**	**False negatives, %**
140	4.0	0.0
160	0.0	0.0
250	0.0	0.1

Table 1: Rate of false positives and false negatives depending on the value of handshake threshold. The threshold is presented for 10 memory accesses.

and the load generator was connected via a 10Gb network. The results are presented in Figure 8.

In line with the previous measurements, Varys reduces the maximum throughput by 19% if the system is configured for a low AEX rate. Otherwise, the AEX rate becomes higher, cache flushing has to happen more frequently and the overhead increases. The higher rate comes from two sources: disabling DynTicks increases the frequency of timer interrupts and disabling interrupt redirection adds exits caused by network interrupts. Finally, the "Over-assignment" line is the throughput of Nginx in the scenario, when we do not dedicate a core exclusively to Nginx and assign another application that competes for the core (in our case, we use *word_count* from Phoenix). Since the Nginx threads are periodically suspended, the cost of the handshake becomes much higher as both threads in a pair have to wait while one of them is suspended.

7.2 Security Evaluation

Violation of trusted reservation. To evaluate how effective Varys is at ensuring trusted reservation (i.e., if a pair of threads is running on the same physical core), we performed an experiment that emulates a time-sliced attack. We launch a dummy Varys-protected application in normal configuration (all threads are correctly paired) and then, at runtime, change affinity of one of the threads. Additionally, to evaluate the rate of false positives, we run the application without the attack. As trusted reservation is implemented via a periodic handshake, the main configuration parameter is the time limit distinguishing cache hits from cache misses.

The results are presented in Table 1. False negatives represent the undetected attacks and false positives—the cases when there was no attack, but a handshake still failed. The results are aggregated over 1000 runs.

As we see, trusted reservation can be considered reliable if the limit is set to 160 ticks of the software timer (§6.2). The fact that we neither have false positives nor false negatives is caused by the difference in timing of L1 and a LLC cache hits. If the threads are on the same core, the handshake will have timing of 10 L1 cache hits. Yet, if they are on different cores, the only shared cache is LLC and all 10 accesses would miss both L1 and L2.

Increased rate of AEX. To evaluate Varys's effectiveness at detecting attacks with high AEX frequencies, we ran a protected application under different system interrupt rates and counted the number of aborts (i.e., detected potential attacks). For the purity of the experiment, the victim was a dummy program that does not introduce additional AEXs on top of the system interrupts. In each of the measurement, we tested several limits on minimal runtime (MRT), inverse of the AEX rate. Similar to the previous experiment, we had 1000 runs.

The results are presented in Table 2. Here, the "Normal rate" is 100Hz (see §5.3); "Low-AEX attack" is 5.5kHz as in the attacks from Wang et al. [45] and Van Bulck et al. [43]; "Common attack" is 10kHz which corresponds to the rate required for cache attacks. We can see that

MRT, IR instructions	**Normal execution**	**Low-AEX attack**	**Common attack**
60M	0.2%	100%	100%
62M	1.2%	100%	100%
64M	10%	100%	100%

Table 2: Varys abort rate depending on the system interrupt rate and on the value of minimum runtime (MRT).

```
for (Set in L1_Cache_Sets):
  for (Very Long):
    for (CLine in CacheLine1..CacheLine8):
      Read(Set, CLine)
```

Figure 9: An example of worst-case victim for a defense mechanism based solely on interrupt frequency.

if we set the threshold on the number of IR instructions between enclave exits to 60 millions, it achieves both low level of false positives (0.2%) and detects all simulated attack attempts.

Residual cache leakage. For small applications (i.e., applications with small or slowly changing working set), cache leakage may persist even after we limit the frequency of enclave exits.

As a worst case, we consider the following application (see Figure 9): it iterates over cache sets, accessing all cache lines in a set for a long time. With such applications, limiting the interrupt frequency will not help, because even a few samples are enough to derive the application state. We use this application to evaluate effectiveness of the cache eviction mechanism proposed in §5.2.

We use a kernel module to launch a time-slicing cache attack on the core running the victim application. The attack delivers an interrupt every 10 ms, and does an L1d cache measurement on all cache sets. We normalize the results into the range of $[0,1]$. Additionally, we disables CPU prefetching both for the victim and attack code to reduce noise. Essentially, it is a powerful kernel-based attack that strives to stay undetected by Varys.

The results of the measurements are on Figure 10a. Without eviction, the attack succeeds and the state of application can be deducted even with a few samples. Then, we apply Varys with L1i and L1d cache eviction to the application (Figure 10b). Even though the amount of information leaked decreases greatly, we can still distinguish some patterns in the heatmap due to residual L2 cache leakage. When we enable L2 eviction in Varys, the results contain no visible information about the victim application (Figure 10b).

8 Hardware Extensions

Many parts of Varys's functionality could be implemented in hardware to improve its efficiency and strengthen the security guarantees. In this section, we propose a few such extensions. We believe that introducing such a functionality would be rather non-intrusive and should not require significant architectural changes.

8.1 Userspace AEX handler

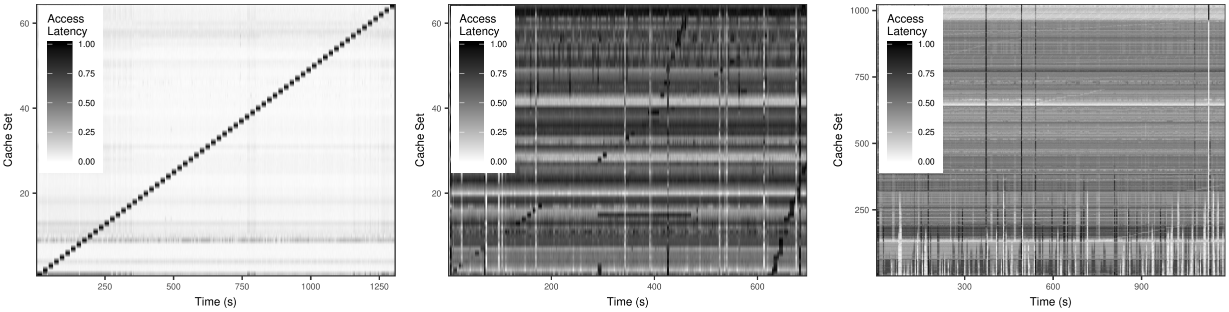
Varys relies on the SGX state saving feature for detection of enclave exits. However, this approach has certain drawbacks: it requires the application to monitor the SSA value, thus increasing the overhead, and it introduces a window of vulnerability (§6.1). An extension to the AEX protocol could solve both of the issues.

Normally during an AEX, the control is passed to the OS exception handler, which further transfers control to the userspace AEX handler, provided by the user. The user AEX handler then executes `ERESUME` instruction, which re-enters the enclave. However, there is no possibility for an in-enclave handler. Our proposed extension adds a hardware triggered callback to the `ERESUME` instruction, specified in the `TCS: TCS.eres_handler`. After each `ERESUME` executed by unprotected code, the enclave is re-entered, and the control is passed to code located at the address `TCS.eres_handler`. To continue executing interrupted in-enclave code, the `ERESUME` handler will execute the `ERESUME` instruction once again, this time, inside the enclave. Note that calling `ERESUME` inside of an enclave is right now not permitted. One difficulty of this extension would be an AEX during the processing of a handler. We would allow recursive calls since handlers could be designed to deal with such recursions.

8.2 Intel CAT extension

Although Intel CAT could be used to prevent concurrent LLC attacks, the OS has complete control over the CAT configuration, which renders the defense ineffective. It can be solved by associating the CAT configuration registers with *version numbers* that are automatically incremented each time the configuration changes. The application could check the version number in the AEX handler after each AEX and thus easily detect the change. In case, no support for AEX handlers is added, the application could perform periodic checks within the enclave instead.

To estimate the potential impact of the extension, we ran an experiment where Nginx was protected by Varys and had a slice of LLC exclusively allocated to it (see Figure 11). As we see, allocating 4 and 2 MB of cache did not cause a significant slowdown for the given workload. The difference in throughput comes mainly from the larger eviction region: Varys had to flush 4 MB instead of 256 KB. However, allocating this large part of the cache can significantly reduce the overall system performance. At the same time, if we try a more modest allocation, we risk causing a much higher rate of cache misses, which is what happened with the 1 MB allocation in our experiment.



(a) No eviction.

(b) L1 eviction.

(c) L2 eviction.

Figure 10: An experiment proving the effectiveness of cache eviction. Without eviction, we can easily see the program behavior. With L1 eviction, the L2 residual leak exposes some information. With L2 eviction, no visible information is exposed. Graphs have different time scales due to different overhead from L1/L2 measurement and presence of eviction mechanism. Color reflects normalized values, with different absolute minimum and maximum values for every graph.

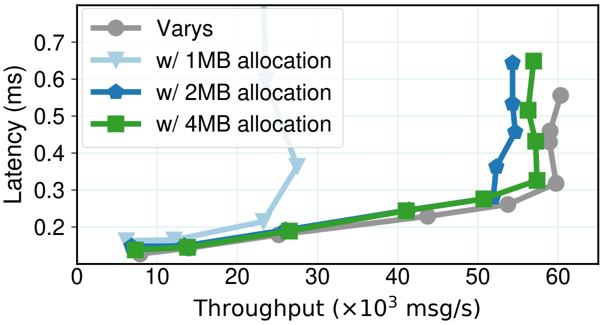


Figure 11: Impact of different cache allocation sizes on throughput and latency of Nginx protected by Varys.

8.3 Trusted HW timer

Since the hardware timer (RDTSC/P instruction) is not available in SGX1, we use a software timer, which wastes a hyperthread. SGX2 is going to introduce the timer, but we cannot rely on it either as privileged software can overwrite and reset its value.

We see two ways of approaching this problem: We may introduce a monotonically increasing read-only timer which could be used as-is. Alternatively, we could introduce a version number that is set to a random value each time the timer is overwritten. To ensure the timer correctness, the application would have to compare the version of this register before and after the measurement.

9 Related Work

The idea of restricting minimal runtime was proposed by Varadarajan et al. [44], although they relied on features of privileged software. Similarly, Déjà Vu [14] relies on measuring execution time of execution paths at run-time.

T-SGX [40] uses Transactional Synchronization Extensions (TSX) to detect and hide page faults from the OS. It protects against page fault attacks, but not page-bit and cache timing attacks. Cloak [22] strives to extend T-SGX guarantees to cache attacks by preloading sensitive data, but requires source code modifications.

Concurrently with our work, an alternative approach to establishing thread co-location was proposed in HyperRace [13]. It uses data races on a shared variable as a way of distinguishing L1 from LLC sharing. Accordingly, it does not require a timer thread.

Zhang et al. [51] and Godfrey at al. [19] employ flushing as a defense against cache attacks, and Cock [15] proposed to used lattice scheduling [16] as an optimization. All of them rely on privileged software.

Among the alternatives, Racoon [37] builds on the idea of oblivious memory [20] and makes enclaves' memory accesses independent of the input by introducing fake accesses, but requires manual changes in code. Dr. SGX [8] automates the obfuscation. Shinde et al. [41] make the accesses deterministic at the page level. Both introduce high overheads (in the range of 3–20×).

10 Conclusion

We presented Varys, an approach to protecting SGX enclaves from side channel attacks. Varys protects from multiple side channels and causes low overheads. Conceptually, Varys protects against side channels by limiting the sharing of core resources like L1 and L2 caches. We have shown that implementing it in software is possible with reasonable overhead. With additional hardware support, we would not only expect a more straightforward implementation of Varys but also lower overhead and protection against a wider range of side channel attacks, including LLC-based ones.

Acknowledgments. We thank our anonymous reviewers for the helpful comments. This work was partly funded by the Federal Ministry of Education and Research of the Federal Republic of Germany (03ZZ0517A, FastCloud) and by Horizon 2020 Research and Innovation Programme (690111, SecureCloud).

References

[1] nginx: The architecture of open source applications. `www.aosabook.org/en/nginx.html`, 2016. Accessed: May, 2018.

[2] ACIIÇMEZ, O. Yet another microarchitectural attack: Exploiting I-cache. In *Workshop on Computer Security Architecture* (2007).

[3] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., GOLTZSCHE, D., EYERS, D., KAPITZA, R., PIETZUCH, P., AND FETZER, C. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[4] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[5] BERNSTEIN, D., LANGE, T., AND SCHWABE, P. The security impact of a new cryptographic library. *Progress in Cryptology–LATINCRYPT 2012* (2012).

[6] BERSHAD, B. N., LEE, D., ROMER, T. H., AND CHEN, J. B. Avoiding conflict misses dynamically in large direct-mapped caches. In *ACM SIGPLAN Notices* (1994).

[7] BIENIA, C., AND LI, K. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)* (2009).

[8] BRASSER, F., CAPKUN, S., DMITRIENKO, A., FRASSETTO, T., KOSTIAINEN, K., MÜLLER, U., AND SADEGHI, A.-R. DR. SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. *arXiv:1709.09917* (2017).

[9] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software Grand Exposure: SGX Cache Attacks Are Practical. *arXiv preprint arXiv:1702.07521* (2017).

[10] BRUMLEY, B. B., AND HAKALA, R. M. Cache-timing template attacks. In *International Conference on the Theory and Application of Cryptology and Information Security* (2009), Springer.

[11] CHE TSAI, C., PORTER, D. E., AND VIJ, M. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017).

[12] CHEN, G., CHEN, S., XIAO, Y., ZHANG, Y., LIN, Z., AND LAI, T. H. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. *arXiv preprint arXiv:1802.09085* (2018).

[13] CHEN, G., WANG, W., CHEN, T., CHEN, S., ZHANG, Y., WANG, X., LAI, T.-H., AND LIN, D. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In *IEEE Symposium on Security and Privacy* (2018).

[14] CHEN, S., REITER, M. K., ZHANG, X., AND ZHANG, Y. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu. In *ASIA CCS '17* (2017).

[15] COCK, D. Practical probability: Applying pGCL to Lattice scheduling. In *Interactive Theorem Proving: 4th International Conference* (2013).

[16] DENNING, D. E. A lattice model of secure information flow. *Communications of the ACM* (1976).

[17] DISSELKOEN, C., KOHLBRENNER, D., PORTER, L., AND TULLSEN, D. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *Usenix Security* (2017).

[18] GE, Q., YAROM, Y., LI, F., AND HEISER, G. Contemporary Processors Are Leaky – and There's Nothing You Can Do About It. *arXiv preprint arXiv:1612.04474* (2016).

[19] GODFREY, M., AND ZULKERNINE, M. A server-side solution to cache-based side-channel attacks in the cloud. In *IEEE Sixth International Conference on Cloud Computing (CLOUD)* (2013).

[20] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* (1996).

[21] GÖTZFRIED, J., ECKERT, M., SCHINZEL, S., AND MÜLLER, T. Cache attacks on Intel SGX. In *European Workshop on System Security (EuroSec)* (2017).

[22] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium (USENIX Security 17)* (2017).

[23] GUERON, S. Intel's new AES instructions for enhanced performance and security. In *Fast Software Encryption: 16th International Workshop* (2009).

[24] HÄHNEL, M., CUI, W., AND PEINADO, M. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017).

[25] INCI, M. S., GULMEZOGLU, B., IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems* (2016).

[26] INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2016.

[27] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *2015 IEEE Symposium on Security and Privacy* (2015).

[28] KESSLER, R. E., AND HILL, M. D. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)* (1992).

[29] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203v1* (2018).

[30] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (2004).

[31] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *HPCA* (2016).

[32] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy* (2015).

[33] MARSHALL, A., HOWARD, M., BUGHER, G., AND HARDEN, B. Security best practices for developing windows azure applications. Microsoft Corp, 2010.

[34] OLEKSENKO, O., KUVAISKII, D., BHATOTIA, P., AND FETZER, C. Fex: A Software Systems Evaluator. In *Proceedings of the 47st International Conference on Dependable Systems & Networks (DSN)* (2017).

[35] ORENBACH, M., LIFSHITS, P., MINKIN, M., AND SILBERSTEIN, M. Eleos: ExitLess OS Services for SGX Enclaves. In *EuroSys* (2017).

[36] PERCIVAL, C. Cache missing for fun and profit. 2005.

[37] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium* (2015).

[38] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)* (2007).

[39] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., and Mangard, S. Malware guard extension: Using SGX to conceal cache attacks. *CoRR abs/1702.08719* (2017).

[40] Shih, M., Lee, S., and Kim, T. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS* (2017).

[41] Shinde, S., Chua, Z. L., Narayanan, V., and Saxena, P. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security - ASIA CCS '16* (2016).

[42] Tromer, E., Osvik, D., and Shamir, A. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology 23*, 1 (2010).

[43] Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., and Strackx, R. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Usenix Security* (2017).

[44] Varadarajan, V., Ristenpart, T., and Swift, M. Scheduler-based defenses against cross-VM side-channels. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014).

[45] Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., and Gunter, C. A. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. *arXiv preprint arXiv:1705.07289* (2017).

[46] Wu, Z., Xu, Z., and Wang, H. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security Symposium* (2012).

[47] Xu, Y., Cui, W., and Peinado, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy* (2015).

[48] Yarom, Y., and Falkner, K. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium* (2014).

[49] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2012), CCS '12.

[50] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014).

[51] Zhang, Y., and Reiter, M. K. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013).

Kernel-Supported Cost-Effective Audit Logging for Causality Tracking

Shiqing Ma[$], Juan Zhai[§], Yonghwi Kwon[$], Kyu Hyung Lee[*], Xiangyu Zhang[$],
Gabriela Ciocarlie[†], Ashish Gehani[†], Vinod Yegneswaran[†], Dongyan Xu[$], Somesh Jha[‡]
[$]*Purdue University,* [§]*Nanjing University,* [*]*University of Georgia,*
[†]*SRI International,* [‡]*University of Wisconsin-Madison*

Abstract

The Linux Audit system is widely used as a causality tracking system in real-world deployments for problem diagnosis and forensic analysis. However, it has poor performance. We perform a comprehensive analysis on the Linux Audit system and find that it suffers from high runtime and storage overheads due to the large volume of redundant events. To address these shortcomings, we propose an in-kernel cache-based online log-reduction system to enable high-performance audit logging. It features a multi-layer caching scheme distributed in various kernel data structures, and uses the caches to detect and suppress redundant events. Our technique is designed to reduce the runtime overhead caused by transferring, processing, and writing logs, as well as the space overhead caused by storing them on disk. Compared to existing log reduction techniques that first generate the huge raw logs before reduction, our technique avoids generating redundant events at the first place. Our experimental results of the prototype KCAL (Kernel-supported Cost-effective Audit Logging) on one-month real-world workloads show that KCAL can reduce the runtime overhead from 40+% to 15-%, and reduce space consumption by 90% on average. KCAL achieves such a large reduction with 4% CPU consumption on average, whereas a state-of-the-art user space log-reduction technique has to occupy a processor with 95+% CPU consumption all the time.

1 Introduction

Understanding system provenance is an important and challenging task, especially in forensic analysis and problem diagnosis. A common approach is to perform operating system-level audit logging, which is one of the core functionalities required in enterprise-level infrastructures. The Linux Audit system is the most widely used audit system. It resides in the kernel, collects information for predefined kernel events, and records them in log files. Following incidents, investigators use automated tools (e.g., ausearch) to analyze audit logs to search for suspicious system objects (e.g., files, sockets) and subjects (e.g., processes), and identify causal dependencies among them. Such information is critical to locating root causes and assessing damages. Then they use such information to hunt for suspicious activities such as policy violations. In practice, the Linux Audit system has been known to have poor performance, and other researchers have been working on improving the Linux Audit system for a long time. Many works [8, 9, 22, 24, 31, 41, 42] proposed enhancement or alternative designs to provide fast logging infrastructures or highly compressed logs. However, existing solutions do not fundamentally solve the high space and runtime overhead problems. And this motivates us to deeply analyze and understand the overhead problems in the Linux Audit framework.

In this paper, we first describe a comprehensive analysis on the Linux Audit system, and show that the runtime and the storage overheads are essentially caused by transferring and processing huge raw logs that contain substantial redundancies. Previous research failed to solve the problems because methods required first generating the redundant logs. Our key idea is *to remove redundancies inside the kernel so that we can prevent the huge raw logs from being generated at the first place.* Inspired by hardware/software cache system designs, we propose KCAL, a kernel-level, cost-effective, memory-cache-based audit logging system. It caches important dependencies and events, and detects redundancy on the fly using the caches. If redundant events indicated by cache hits are detected, they are immediately discarded. Only events that introduce new system objects/subjects or new dependencies are retained. Dependency caches and event caches are distributed in individual kernel data structures. The caches are carefully designed such that the kernel memory consumption is kept reasonably low, avoiding perturbation of normal kernel functionality. In summary, in this paper, we make the following contributions:

- We describe a comprehensive analysis on the Linux Audit system, which revealed that the root cause of its high runtime and storage overheads is the need to transfer, process, and store the huge raw log, and identify this can be solved by removing the redundant events.
- We propose a kernel-level, cache-based, log-reduction system. The key idea is to prevent the kernel from generating redundant raw logs in the first place. The design features a multi-layered and distributed cache scheme that leverages the autonomous execution sub-structures (i.e., units) in individual processes (e.g., sub-executions serving individual requests in Apache), and indexes largely scattered syscall events belonging to the same object.
- We built a prototype KCAL based on the Linux Audit system. Our experimental results showed that KCAL is capable of reducing the runtime overhead from 40+% to 15-%, log files by 90+%, and it does not introduce significant memory pressure on the existing kernel. The comparison with the state-of-the-art, user-space log-reduction technique ProTracer [24] shows that ProTracer fully occupies an idle processor with 95% constant CPU consumption whereas KCAL only requires 4% CPU consumption on average.

2 Motivation and Related Works

2.1 Audit Logging Systems

There are many existing audit logging systems [2, 5, 7, 24, 28, 31] from commercial companies and research communities. Prior works [13, 14, 16, 35, 37, 43] proposed many different general logging infrastructures. Some of them [11, 27, 28, 34] monitor the whole file system at the `inode` level, while others [31, 36] leverage the Linux Security Module (LSM) to monitor operations on kernel data structures. Many of the techniques [12, 17, 18, 19, 20] use record-and-replay techniques to record system wide events for system replay. They require logging of syscalls including the concrete values such as the content of files or packets. Hence, they tend to be expensive and are mostly used in single application execution. Bates *et al.* provide a general and secure framework for writing a provenance system at the operating system level. Among these provenance systems, the Linux Audit framework [2] is the most practical and widely used. The framework provides a general logging infrastructure that allows the integration of plugins to enhance the system. As such, it is widely used and has been adopted by many other research projects and real-world products [3, 4, 6].

Linux Audit Architecture. Figure 1 shows the architecture of the Linux Audit framework. It contains a few user-space utilities (brown boxes) and a kernel component. The kernel component contains a number of filters (blue circles). Based on the execution order, i.e., before/during/after the syscall processing logic, the filters are named `User/Task/Exit`, respectively. The `Exclude` filter defines exceptions to the filtering rules; namely, any syscall that falls into the `Exclude` category will not be filtered. The `auditctl` program helps administrators manage filters. If these filters determine that a syscall needs to be recorded, the kernel component sends the information to the user-space daemon program `auditd` through the `Netlink` device. `Auditd` collects syscall records and writes them to the log file.

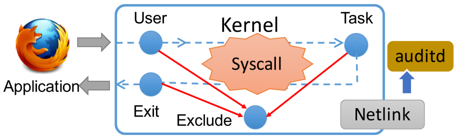
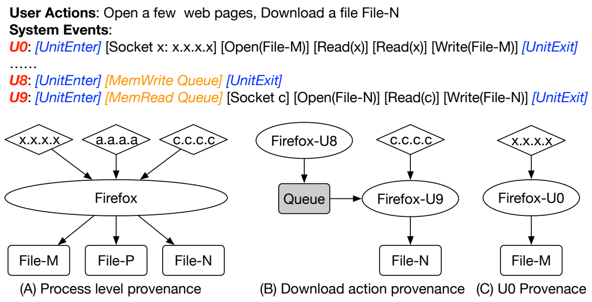


Figure 1: The audit framework architecture

State-of-the-Art Causality Analysis. An important feature of an audit logging system is the dependency analysis support. As demonstrated by previous researchers [21,23], the Linux Audit system suffers from the *dependency explosion* problem because of the large number of fan-outs in process-level analysis. Process execution partitioning techniques [21, 23] were proposed to enable fine-grained dependency analysis in audit logging, and to help remove redundant log information. They partition process executions into *execution units*. Each execution unit is a part of the whole process execution serving a specific task. MPI [23] partitions process execution based on user-defined tasks, e.g., individual tabs in Firefox. BEEP [21] partitions process execution based on event-handling loop, namely, and an execution unit is essentially an iteration of the event handling loop. Execution units are considered largely autonomous. Therefore, *an output syscall event in a unit is considered only dependent on the preceding input events within the same unit unless there are dependencies across units (e.g., through in-memory data structures)*. In contrast, Linux Audit considers that an output event depends on all the preceding events in the same process, causing numerous bogus dependencies [14,18,20,21]. An execution unit is delimited by a special `UnitEnter` event indicating the start of the unit, and a `UnitExit` event denoting the end in these systems. An execution unit may depend on another through variable reads/writes. Such variables/data-structures are treated as Inter-Process Communication (IPC) objects, and exposed to the audit system via the `MemWrite` and `MemRead` syscall events [21, 23].

Figure 2 shows an example of using Firefox to open webpages and download a file (File-N). It also shows the simplified log events. In each line, we show the events



User Actions: Open a few web pages, Download a file File-N
System Events:
U0: *[UnitEnter]* [Socket x: x.x.x.x] [Open(File-M)] [Read(x)] [Read(x)] [Write(File-M)] *[UnitExit]*
......
U8: *[UnitEnter]* *[MemWrite Queue]* *[UnitExit]*
U9: *[UnitEnter]* *[MemRead Queue]* [Socket c] [Open(File-N)] [Read(c)] [Write(File-N)] *[UnitExit]*

(A) Process level provenance (B) Download action provenance (C) U0 Provenace

Figure 2: Comparison of different dependency granularity

that belong to a unit marked with the unit ID. Without unit information, we will get a graph shown in (A). The file object File-M (transitively) depends on all the socket read by Firefox before the write to the file, which introduces many bogus dependencies. With partitioning, the events are properly grouped. For example, the first unit, *U0* represents a unit for opening a web page. It creates a socket, fetches a page, stores it, and then renders it on screen. The dependency relationship is shown in (C). The multiple page loading tasks are separated to units, and the resulting provenance graph is accurate. Downloading in Firefox causes explicit dependencies between units (*U8*, *U9*). *U8* first inserts the download request to a queue, and then *U9* fetches it from the queue and downloads the file. These units are connected through `MemWrite`/`MemRead` events as shown in (B).

2.2 Linux Audit System Performance

To motivate our technique, we perform a few experiments to measure the overhead of Linux Audit and explain the limitations of existing overhead reduction techniques. We run 20 virtual machines with Ubuntu 14.04 as the guest OS on the Kernel-based virtual Machine (KVM) platform, and each virtual machine has two cores and 4 GB main memory. The machines are classified into two categories: servers running server programs (e.g., HTTP server Apache, FTP server ProFTPd), and clients running client programs (e.g., Firefox and Vim) for daily use. Each group has 10 virtual machines.

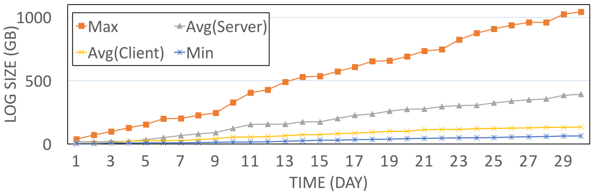


Figure 3: The audit framework log sizes growth in 30 days

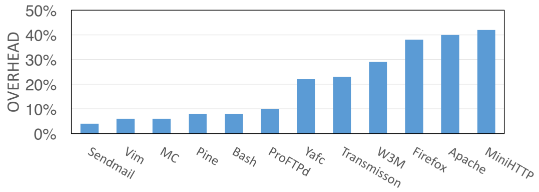


Figure 4: The audit framework runtime overhead

Figure 3 shows the log size growth along time. We configure the audit system to only record 60 provenance-related syscalls [24, 31]. These system calls are related to process creation/termination, file/socket creation/read-/write/deletion and IPCs and so on. Observe that in the worst case, a machine generates 1100+GB log in 30 days. Even in the best case, 60 GB log is generated within 30 days. On average, a server machine can generate about 130GB data per day, whereas a client machine generates about 5GB data per day. The data is also consistent with previous research [22, 41]. Such a large volume of data causes many problems. First, it is expensive to store or transmit log files. By default, the audit log is stored on the local disk and consumes substantial storage space. It can be sent to external servers for storage and inspection, but this incurs runtime overhead, network traffic, and maintenance efforts on servers. Second, processing such large files can be extremely challenging. It may take hours to days to answer a provenance query as it requires searching through log files in the size of GBs to TBs. The situation becomes worse in the enterprise environment, where there are hundreds to thousands of inter-connected machines, which increases the problems associated with storing, correlating, and processing audit logs. Compressing the log is one way to reduce the storage overhead, but causes more runtime overhead for compressing/decompressing the logs to investigate an attack. Zhang *et al.* [41] also demonstrated that in an enterprise environment, using databases to store the logs is also very challenging in such scenarios.

Figure 4 shows the runtime overhead (caused by Linux Audit) for a few programs including both server programs like Apache and client programs like Vim. We leverage existing workloads to test the performance if possible. For programs that support batch mode (e.g., Vim), we write scripts to test the performance. Some of the programs generate frequent system calls (e.g., Apache), and naturally cause higher runtime overhead. As we can see, the overhead for some programs like Vim is tolerable. But for I/O intensive programs such as browsers and server programs, the overhead can be rather high.

Understanding the Overheads. The Linux Audit framework has three parts: the kernel part (filtering rules etc.), the `Netlink` data transmission channel, and the user space logger (i.e., `auditd`). Figure 5 shows the runtime overhead of each component. Similar observations are made on both Hard Disk Drives (HDDs) and Solid State

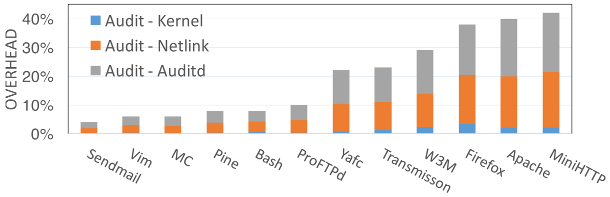


Figure 5: Audit framework runtime overhead

Drives (SDDs). The graph tells us the kernel filters are relatively lightweight, and the other two parts, `Netlink` and `auditd`, are the dominant factors of the overhead. `Netlink` provides a socket-like channel for transmitting data, and `auditd` is responsible for writing the log data to disk. The major factor that affects the time spent on these two components is the size of log data that needs to be processed (transferred/written). Considering the amount of data we need to handle (Figure 3), it is understandable these two parts dominate the overhead. As such, we can say *the root cause of both runtime and storage overheads is the large amount of data generated by the Linux Audit framework*.

2.3 Log Redundancy

Previous works addressed the storage overhead problem by shrinking the log size. Most of them [10, 15, 25, 29, 30, 32, 33, 38, 39, 40] generated the dependency graph first, and then used various graph visualization or compression algorithms to help causality analysis. These techniques ignore the importance of reducing the redundancy of audit logs, and cannot solve the runtime overhead problem caused by such redundancy.

Existing Linux Audit generates highly redundant logs. Based on our analysis (see §4), over 89% log entries are redundant. Previous research [22, 24, 41] has also presented similar observations. Thus pruning the log could improve the performance of the Linux Audit system. Some existing works [8, 41] suggest removing redundancy by various analysis techniques, e.g., rule-based filtering. However, this requires human effort to create and maintain the rules. ProTracer [24] leverages execution partitioning for log reduction. It has a kernel module, which simply receives syscall events, filters them, and then sends the remaining event records including unit-related events to the user-space daemon, which consists of multiple processes. These processes run in parallel to remove redundant events. The ProTracer views system objects as taints and monitor their propagation during execution by performing syscall level taint analysis while processing the log. Each unit/object is associated with a taint set denoting the set of data sources that it depends on. The causalities denoted by the taint sets (instead of individual events) are emitted to the log. Therefore events leading to the same taint set are essentially reduced.

All these techniques first generate the full-fledged log and then reduce it. It is the huge raw log that causes the substantial overhead. These techniques cannot be applied in the kernel space because it has rather limited resources that prevents loading and processing huge raw logs. For instance, the parallel (tainting-based) processing required by ProTracer cannot be ported to the kernel space due to its high CPU consumption (See data in §4). An ideal solution is to *prevent redundant log entries from being generated by the kernel in the first place*. This is the motivation behind KCAL, a kernel-supported log cache and reduction system.

3 Design

3.1 Overview

We propose a cache-based, cost-effective audit logging system inside the kernel called KCAL. It leverages execution partitioning and is orthogonal to the underlying partitioning scheme. Any partitioning scheme [21, 23] that generates unit boundary syscalls and cross-unit memory dependency events can be seamlessly integrated with KCAL, and we use BEEP. Upon a new syscall event, KCAL determines if there is a cache hit, which means the new event reveals the same causal information as some event(s) that have been recorded before and hence can be safely discarded. Since the cache is positioned at the kernel, redundant log events are prevented from being generated in the first place, leading to highly succinct raw logs without any information loss. KCAL is not a monolithic caching system like traditional memory caches because different subjects/objects have diverse life times and various numbers of associated syscall events distributed in their life spans. Due to the nature of audit logging, we cannot be certain if events belonging to a subject/object are redundant before it is closed or terminated. A monolithic cache design would require complex data structure support for indexing and removing sparse and highly distributed log events. Therefore, we propose a distributed cache design so each process/object (e.g., a file) has its own cache storing associated events, and these caches are encapsulated as part of the kernel data structures. Figure 6 shows the overall architecture of KCAL.

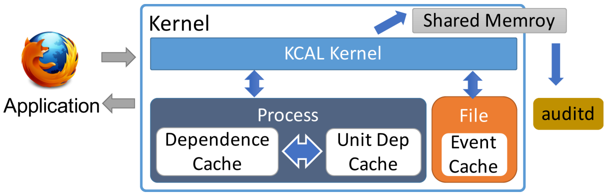


Figure 6: Overview of KCAL architecture

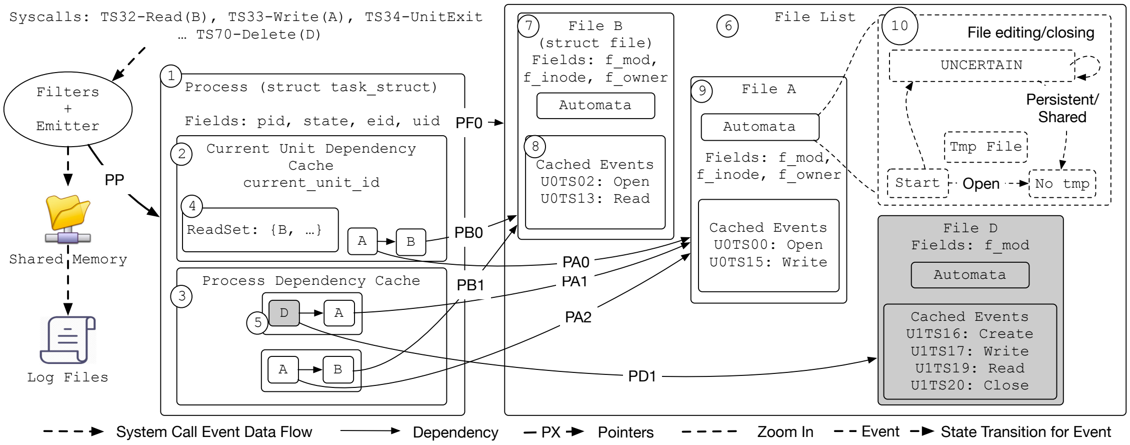


Figure 7: Overview of KCAL

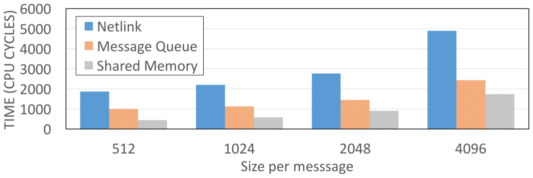


Figure 8: Performance of 3 data channels (kernel to user-space)

First, we enhance the Linux Audit module with an online cache-based log-reduction algorithm, and modify the kernel data structures for processes and objects (e.g., files and sockets) to insert caches. Second, we use shared memory instead of `Netlink` as the transfer channel.

In-kernel Architecture and Workflow. Figure 7 shows a simplified view of the kernel part of KCAL. The first modification is in the `task_struct` data structure (①), which stores process specific information such as the *pid*. We add more pointer fields. The first one is a pointer to a unit dependency cache (box ②). The cache uses a *ReadSet* to store the objects that have been read by the current unit (box ④), and also maintains the detected dependencies in the current unit (e.g., $A \rightarrow B$ shown inside ②). Each object in the cache also has a pointer (e.g., PA0 and PB0) to the corresponding kernel data structure instance such as a `File` structure. The second pointer points to a process-level dependency cache (box ③), which stores the dependencies detected in this process (box ⑤) by aggregating the unique dependencies from individual units. The unit cache is needed for in-unit redundancy and the process cache is for cross-unit redundancy.

We also enhance the kernel data structures representing objects (resources). For example, we enhance the `File` data structure that contains file-specific information, such as its inode, by adding two pointers. The first one points to a cache that stores the syscall events operating on the object with timestamps (box ⑧). Redundant events are removed at the unit/process level before being added to the object cache. We do not directly send these events to the user space but rather cache them because all the events in the object cache may be deemed redundant if the resource is determined as temporary. More details will be discussed in §3.3. The second pointer points to an automaton used to detect if the resource is temporary (box ⑨). Box ⑩ shows the states and the transitions. Details will be discussed in §3.3. KCAL does not cause any compatibility issues as it does not change the meanings of existing fields in these data structures. More importantly, our method is general, and one could easily use stand-alone hash tables that map a process/object to its auxiliary data structures and avoid touching any kernel data structures. As we will show in §4, although KCAL is mainly kernel based, its perturbation to the normal kernel functionalities is negligible due to its small memory footprint and limited instrumentation inside the kernel.

The Linux Audit module is enhanced with an on-the-fly reduction algorithm that interacts with the caches to determine if an event is redundant. When a syscall event occurs, it first goes through the filters. Non-provenance related syscalls like time-related system calls are filtered out. The remaining syscalls (i.e., reads/writes) are passed to the reduction component. This component checks if there is a cache hit for the dependency represented by the event. Note the caches are accessible through the `current` variable, which points to the `task_struct` of the current process that contains direct or transitive pointers to multiple layers of caches. If hit, the event is safely discarded. Otherwise the dependence is inserted to the dependence cache, and the event is inserted to the event cache of the object that is being operated on. Eventually, non-redundant events will be emitted to the shared memory and saved to the disk by the user-space component.

Transfer Channels. `Netlink` provides a socket-like communication method between the kernel space and the user space and was widely adopted by SELinux as it provides a simpler interface and better performance as compared with its competitors (printk, ioctl etc.). We compare three general ways of transferring bulk data from the kernel space to the user space: `Netlink`, message queues, and shared memory. Figure 8 shows the performance comparison of these channels. The X-axis represents the size of each message. We use four configurations: 512, 1024, 2048, and 4096 bytes. For each message size, we generate 10,000 random messages and perform the experiments 10 times. The Y-axis is the performance measured by the average time (CPU cycles) used to transfer one message. Shared memory has the best performance. In the past, due to the memory size limits made it practical to use shared memory as the transfer channel as it requires reserving a memory pool, but this is no longer a problem in modern computers.

3.2 Redundancy in the Linux Audit Log

Our definition of *redundancy* is with respect to the attack investigation, which is based on a causal graph according to the latest *Open Provenance Model* (OPM) [26]. OPM standardizes the forensic analysis procedure and is the most widely adopted provenance model. A causal graph is generated by first starting from a given subject or object (e.g., a suspicious file) and then performing *forward/backward* traversal along dependencies to find all the reachable subjects and objects. Backward traversal is used when the inspector wants to trace back the root cause of an attack starting from some observed symptom. In contrast, forward traversal is used when the inspector has already identified the root cause and now wants to understand the damage caused by the attack. It starts with the root cause and finds all the affected subjects/objects. In this context, *we consider an event redundant if the derived causal graph contains the same dependency information with and without the event.* That is, we can reach the same set of objects and subjects with and without the event. As such, an event is redundant if it leads to some dependency that was induced in a previous unit. This is because the previously recorded events and the entailed dependence render the same reachability. Events denoting the same dependency may be in the same or different execution units, and they are referred as in-unit redundancy and cross-unit redundancy, respectively.

Another type of redundancy is what we call *temporary files*. We define the term *temporary file* from the provenance analysis perspective. A *temporary file* is a file that is created, edited, and deleted by the same owner and during its whole life cycle; the file is not “shared” with any other process. These files only temporarily exist in the system and are used internally by applications. For example, editors with auto-saving features often use a temporary file to keep the newly edited contents to support recovery. These files are deleted when the user saves the file explicitly. Another example is that browsers like Firefox use a large number of temporary files to store downloaded web elements. The browser regularly removes such files to save space. Temporary files do not lead to useful forensic information as they do not have interactions with other system objects or subjects and can be removed. Many programs use temporary files because they need to save a large amount of contents locally, and memory is not sufficient for them to do so. From the provenance point of view, the temporary files are a part of the program execution just like the runtime variables in memory. As all the information source and sink points are the same, it can be viewed part of the program execution.

Our definition of *temporal files* is different from that of traditional temporary files that usually refers to the files in the `tmp` file system. Many of these files can interact with other processes and generate new provenance information, and thus the corresponding events are not redundant according to our design. For example, Firefox has one `open with` option for many types of files such as torrent files in its download dialog. It will first download the selected file (e.g., one torrent file), and then open the file using the system default program for this file type. In this case, the downloaded file is stored in the `tmp` file system, but it will be kept in our design as it is read by another process, which is new provenance information.

Therefore, KCAL guarantees the information completeness from the provenance graph point of view. Namely, the log files before and after reduction will output the same provenance graph for the same provenance query. As a provenance tracking system, it does not guarantee information completeness for other audit purposes such as the total number of syscalls in a time range.

3.3 Redundancy Reduction

In this section, we explain the details of the KCAL design. There are three important design choices: *a two-layer dependency cache (the unit layer and the process layer) for a process*, *a distributed event cache for objects*, and methods for *handling cross-unit memory dependencies.*

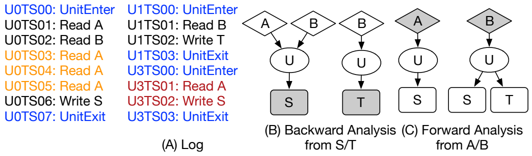
Two-layer Dependency Cache For Process. As shown in Figure 7, we cache two-layer dependencies for a process, the dependencies detected in the current unit (box ②) and those in the whole process (box ③). The former is to remove in-unit redundancy, and the latter is to reduce cross-unit redundancy.

- *In-unit Redundancy.* Read/write syscalls use buffers with limited sizes to transfer data. To load/edit a file larger than the buffers, an application has to issue a sequence of read/write syscalls. For example, Vim reads a file piece by piece and adds the pieces to the in-memory content tree. This produces a sequence of events in the audit log (without reduction) containing tens to hundreds of read syscalls on the target file in the same execution unit. These syscalls denote the same dependency and are redundant. Such redundancy is detected by the unit cache in KCAL. KCAL only keeps one instance of them. At the first read event, the object is added to the `ReadSet`. If it is already in the `ReadSet`, the event is simply discarded. When a write event happens, it is considered dependent on all the objects in the `ReadSet` as the information from these objects can affect the content it writes. KCAL checks if these dependencies are present in the unit-dependency cache. If not, it adds the write event and the read events of objects in the unit cache to the event caches of the corresponding objects. Otherwise, the event is discarded.

- *Cross-unit Redundancy.* Processes usually perform repeated actions on the same system objects. Some of them are because of repeated user actions. For example, editors usually work on a few files for a long time with repeated editing operations. And some of them are due to built-in application functionalities. For example, Firefox writes to the `recovery.js` file every 15 seconds (through a unit) to support purpose. As a result, the same dependency can appear in the log file across different units repeatedly, leading to *cross-unit redundancy*.

An example in Figure 9 on the left-hand-side, (A) shows the simplified log entries. There are three units. Unit 0 (`U0`) reads File-A, File-B, and writes File-S; `U1` reads File-B and writes File-T; and `U3` reads File-A and writes File-S. The blue entries are the `UnitEnter`/`UnitExit` events. The yellow entries are the in-unit redundant events. In particular, `U0` keeps loading contents from File-A. Events `U0TS03` to `U0TS05` all represent the same action, and are redundant. The red entries are the cross-unit redundant events. In this case, the causal dependency between File-A and File-S in `U3` is already detected in `U0`, and hence is redundant. The graphs in (B) show the generated backward analysis graphs starting from File-S and File-T (in gray), and the graphs in (C) represent the generated forward analysis graphs starting from File-A and File-B (in gray). Events `U3TS01` and `U3TS02` do not induce any new dependencies and removing them does not affect the reachable objects and subjects in both forward and backward analyses. Without the execution partitioning, File-T would depend on File-A because the process loads File-A before writing to File-T. As shown later in §4, our reduced logs generate the same casual graphs as using the original BEEP logs (with redundancy reduction).

Distributed-event Cache. In KCAL, dependencies are cached in processes and syscall events are cached in objects. KCAL features a distributed-event cache mechanism in which each object caches the syscall events that operate on the object. They are not transferred to the user space for storage until the object is no longer needed or the process terminates. This is to handle the substantial redundancy caused by temporary files (defined in §3.2).

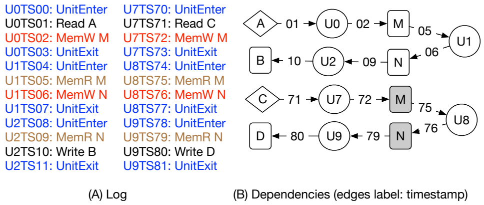


```
U0TS00: UnitEnter   U1TS00: UnitEnter
U0TS01: Read A      U1TS01: Read B
U0TS02: Read B      U1TS02: Write T
U0TS03: Read A      U1TS03: UnitExit
U0TS04: Read A      U3TS00: UnitEnter
U0TS05: Read A      U3TS01: Read A
U0TS06: Write S     U3TS02: Write S
U0TS07: UnitExit    U3TS03: UnitExit
```

(A) Log

(B) Backward Analysis from S/T (C) Forward Analysis from A/B

Figure 9: In-unit and cross-unit redundancy removal example



```
U0TS00: UnitEnter   U7TS70: UnitEnter
U0TS01: Read A      U7TS71: Read C
U0TS02: MemW M      U7TS72: MemW M
U0TS03: UnitExit    U7TS73: UnitExit
U1TS04: UnitEnter   U8TS74: UnitEnter
U1TS05: MemR M      U8TS75: MemR M
U1TS06: MemW N      U8TS76: MemW N
U1TS07: UnitExit    U8TS77: UnitExit
U2TS08: UnitEnter   U9TS78: UnitEnter
U2TS09: MemR N      U9TS79: MemR N
U2TS10: Write B     U9TS80: Write D
U2TS11: UnitExit    U9TS81: UnitExit
```

(A) Log

(B) Dependencies (edges label: timestamp)

Figure 10: Cross-unit memory causality example

We detect temporary files using the automaton shown in box ⑩ in Figure 7. Each `File` data structure maintains its own state. At first, when a file is opened by the owner process, KCAL checks the creator of the file. If the file exists and was created by another process, it is marked as a non-temporary file. Otherwise, it can potentially be a temporary file (i.e., the *UNCERTAIN* state). Normal file editing operations by the owner such as read/write/close do not change the state of the file. Any operation from a different process indicates that information propagates beyond the current process through the file, and hence the file must not be temporary. If the file is deleted by its owner without being read/edited by other processes, it is temporary. If the file is not deleted by its owner process (and hence is persistent), it is not temporary.

As KCAL cannot be certain if a file is temporary or not until the file is deleted, edited by other processes, or the owner process terminates, it buffers all the events for a file it created (after redundancy reduction) in the cache associated with the file until either condition is satisfied. Then, KCAL discards all the events in the cache or emits them to the user space. The emission order of events may be different from the temporal order due to the distributed caching. It is not problematic, however, as all events have global time stamps.

Cross-unit Memory Causality. As mentioned earlier, there may be dependencies across units caused by variables or data structures. For example, in Vim's built-in clipboard, a piece of memory (known as *registers*) is used to support copy/cut-and-paste operation across units.

Existing execution partitioning schemes generate special syscalls `MemWrite`/`MemRead` to denote the write/read operations on cross-unit, dependency-inducing variables, respectively. The nature of these memory operations is very similar to file reads and writes. Hence, KCAL models these events in a similar way. Particularly, each unique memory location is considered as a separate object. The difference is we do not remove events that cause the same memory dependencies across units. Instead, KCAL treats the memory object as a new object each time it is redefined. This is because each memory write to a location is considered as a complete redefinition of the memory object, which is different from a file write. For such an object, each read only depends on the latest write.

For example, in the syscall sequence in Figure 10, unit `U1` receives a request through the memory queue from `U0` at location `M` and then forwards another request to `U2` through a different memory location `N`. KCAL detects a dependency from `N` to `M`. Later, the same procedure happens again and hence the same dependence is detected inside unit `U8`. The same memory locations are observed due to memory reuse, and we cannot unify the multiple instances of the memory locations and throw away the memory events in `U8`. Otherwise, bogus dependencies would be introduced. In the shown example, `D` only depends on `C`. If the dependency introduced by `U8` is removed and the two `M` nodes are unified, `D` would depend on `{C,A}`. In KCAL, the variables `M, N` associated with `U8` and the ones associated with `U1` are treated as a new set of system objects even though they are using the same memory addresses. KCAL leverages existing execution partitioning techniques and existing execution partitioning techniques only instrument a very small number of memory operations through sophisticated analysis [21,23]; and hence, the number of memory events generated at run time is small.

3.4 Implementation and Discussion

KCAL is implemented on the long-supported Linux kernel 3.19 and the Linux Audit framework 2.3. By default, each system object cache size is 32 events. The number of dependencies a process can cache is capped at 256, and the number of dependencies a unit can cache is 8. These values are configurable in KCAL. If the cache is full, and we use the Least Recently Used (LRU) cache replacement policy to evict entries. It is important to note *the consequence of cache eviction is merely that some redundancy cannot be removed. It does not affect information completeness*. The study of the effect of various cache sizes can be found in §4.

KCAL is a provenance tracking system built on top of the Linux Audit framework. The audit log message format is still the same. This makes it compatible with existing audit log processing tools such as `aureport` and `ausearch`. On the other side, the generated messages are for provenance tracking only, and the number of such messages is significantly reduced. This will affect audit tools that calculate the sysall frequencies or concretely analyze individual syscalls such as `aureport`. Also, KCAL modifies the Linux kernel source code including many data structures. As a result, porting it to other kernel versions requires extra human effort. We also port our prototype from Linux kernel 3.19 to kernel 3.2, and most of the patches can be directly applied. The new modification is less than 10 lines. We expect that the porting efforts will be limited as long as the kernel data structure change is not significant. KCAL also depends on existing execution partitioning techniques such as BEEP [21] or MPI [23]. Without the execution partitioning support, cross-unit redundancies cannot be removed, which affects the reduction effectiveness (see §4).

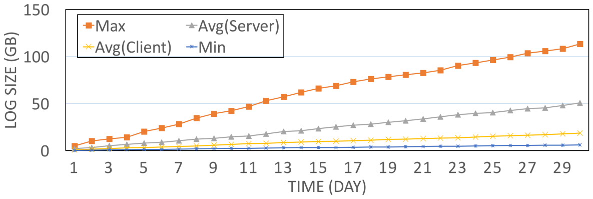


Figure 11: The space overhead of KCAL in a month

4 Evaluation

We evaluate our prototype to answer the following research questions (RQ):
RQ1: How efficient is KCAL? (§4.1)
RQ2: Can KCAL remove the redundancy while keeping the accuracy of the forensic analysis? (§4.2, §4.3)
RQ3: What are the rationales of our design choices, and what are the benefits? (§4.4)

4.1 KCAL Performance

Space Overhead. The space overhead is shown in Figure 11. The experimental environments and workloads are the same with the ones in §2. The orange shows the growth of log size for the machine that has the maximum size. In our case, the log file is less than 120GB after 30 days, while the old log size was almost 1TB without our technique (Figure 3). The gray line represents the average log size for the server machines, and the yellow line shows the average log size for the client machines. Compared with the original audit log (Figure 4), the log size is less than 10%. The workloads also include many applications that do not have the execution partitioning instrumentation, and thus do not benefit from log reduction. The blue lines show the log size of the machine that has the minimal log size. The log now is only about 6GB for 30 days. This shows that KCAL generates much smaller log files than the Linux Audit system.

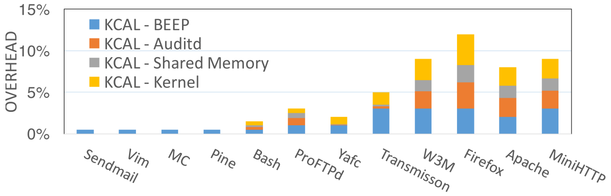


Figure 12: The KCAL runtime overhead analysis

Runtime Overhead. We used the same configuration and the same set of applications with the experiments used in §2 to measure the runtime overhead. Figure 12 shows the results. The bottom bars show the runtime overhead caused by the instrumentation, and the upper bars show the overhead caused by KCAL. For most client programs, the overhead caused by KCAL is less than 1%. The overhead for server programs is about 5% to 10%. This is because a server program needs to serve many clients at the same time, causing a large number of dependencies. Firefox has the most significant overhead, about 15%. This is because Firefox dynamically creates and terminates hundreds of threads, and uses many sockets and files for DNS queries, browsing history, cache, page preferences, and so on. It generates more events within the same duration compared with other applications, leading to higher overhead.

4.2 KCAL Log Reduction Effects

Table 1 summarizes the effects of using our log-reduction algorithm. The first two columns show the experimental environment. The third column shows the number of raw audit logs. We also present the number of log events and the corresponding percentage of in-unit redundancy (columns 4-5), cross-unit redundancy (columns 6-7), and temporary files (columns 8-9). The last two columns show the number of log entries and the percentages after reduction. We first ran the system on five machines for one month, collected the numbers (rows 3-7), and calculated the average (row 8, in gray). For different settings and runs, the reduction effects are different. Some of them have substantial in-unit redundancy (machines 1,2) while others have more cross-unit dependency (machines 4,5). Overall, KCAL keeps only 8% to 14% of the original logs, and on average the number is 11%. We also collected the reduction effects for some representative applications. The results are shown in rows 9-20 in Table 1. For different programs, the effects of the algorithm are significantly different. For example, most server programs do not have temporary files. On the contrary, browsers like Firefox use temporary files a lot. Server programs, especially FTP servers, need to read large files, and generate a huge number of in-unit redundant events, while this is not true for most client programs. For many programs like Vim, the dependency relationships are simple because they work on a limited number of system objects, and we can reduce the events to a very small number. For other programs like Bash, most of their events are related to process manipulations, which cannot be reduced. Thus, most of the logged events are kept. These process-related events will not be cached as they cannot be reduced, and they will not flood the cache. KCAL directly generates such reduced logs from the kernel, leading to substantial savings in raw log transfer from the kernel to the user space and in log processing compared to existing user-space reduction techniques [22, 24, 41].

4.3 Support for forensic analysis

We also performed a few experiments to verify the correctness of our log-reduction algorithm and the benefit for forensic analysis. We reproduced the attack cases used by previous research works [21, 23] and compared the generated graphs and the analysis time. In another set of experiments, we randomly selected 100 objects, and performed backward analysis to identify all of its dependencies. The results are summarized in Table 2. We show the number of nodes and edges in the graphs, the size of the log file, and the analysis time spent using the Linux Audit log, BEEP log, and KCAL log measured by log size. Note that BEEP logs are usually 10-30% larger than the Linux Audit logs due to the additional unit-related events. The last row shows the average number of nodes and edges, the average size of the log files, and the average analysis time for the randomly selected objects. We manually inspected and compared the graphs. The results show that all generated graphs contain the needed and complete information. The graphs generated by the Linux Audit framework usually contain redundant nodes/edges (representing wrong dependencies), whereas graphs generated by the other two methods generated the same graphs. This shows our reduction algorithm is lossless. Because of the complex dependency relationships, it takes far longer time to perform the analysis on the Audit log. BEEP benefits from simpler dependency relationships, but it spends more time inspecting the large number of log entries and checking and updating the dependency sets for each event including many redundant operations. The KCAL log provides simplified and accurate provenance information, enabling faster forensic analysis.

4.4 Understanding KCAL

Cache Behavior. Table 3 shows summarized data for cache behaviors. It shows the name of applications (column-1), the average/maximum number of dependencies in unit cache (column-2), the average/maximum num-

Table 1: KCAL log reduction effects

	Scenario	Audit (#logs)	In-Unit Redundancy		Cross-Unit Redundancy		Temporary Files		KCAL	
			#logs	(%)	#logs	(%)	#logs	(%)	#logs	(%)
Monthly Execution	Machine 1	62,384,284	42,887,843	69%	4,594,385	7%	9,827,394	16%	5,074,662	8%
	Machine 2	137,121,400	97,384,284	71%	13,428,384	10%	12,398,283	9%	13,910,449	10%
	Machine 3	152,385,284	85,727,385	56%	15,228,384	10%	32,299,384	21%	19,130,131	13%
	Machine 4	87,837,384	18,395,394	21%	40,293,293	46%	20,923,283	24%	8,225,414	9%
	Machine 5	93,284,284	27,485,743	29%	40,293,842	43%	12,238,482	13%	13,266,217	14%
	Average	106,602,527	54,376,130	51%	22,767,658	21%	17,537,365	16%	11,921,375	11%
Apps	Firefox	6,284,385	1,128,384	18%	3,238,478	52%	1,248,284	20%	669,239	11%
	Apache	8,942,845	4,829,423	54%	2,684,284	30%	0	0%	1,429,138	16%
	Sendmail	63,284	32,493	51%	12,284	19%	16,293	26%	2,214	3%
	Vim	123,485	36,827	30%	52,284	42%	33,235	27%	1,139	1%
	MC	83,495	16,283	20%	21,384	26%	2,942	4%	42,886	51%
	Bash	20,495	2,342	11%	0	0%	0	0%	18,153	89%
	Pine	10,294	1,023	10%	8,348	81%	494	5%	429	4%
	ProFTPd	3,485,924	3,128,385	90%	100,242	3%	0	0%	257,297	7%
	Yafc	924,395	801,384	87%	39,274	4%	0	0%	83,737	9%
	Transmission	88,384	5,394	6%	80,283	91%	1,482	2%	1,225	1%
	W3M	2,485,395	423,242	17%	1,024,385	41%	743,284	30%	294,484	12%
	MiniHTTP	98,285	78,283	80%	12,384	13%	0	0%	7,618	8%

Table 2: Forensic analysis cases

Cases	Audit				BEEP				KCAL			
	#Node	#Edge	Size(MB)	Time(s)	#Node	#Edge	Size(MB)	Time(s)	#Node	#Edge	Size(MB)	Time(s)
Phishing	317	354	1905	2234	18	23	2096	142	18	23	168	16
Intrusion	860	2135	1626	30864	5	4	1888	162	5	4	226	18
InfoTheft	51	51	1148	823	7	6	1286	92	7	6	154	10
Random	412	683	2345	3349	14	32	1532	122	14	32	169	14

Table 3: KCAL cache summary (avg/max)

Application	#Deps/Unit	#Deps/Pr	#Events/Obj
Firefox	0.8/4	123/256	7.4/18
Apache	1.8/4	52/69	8.6/12
Sendmail	0.6/3	7/12	8.2/16
Vim	0.2/2	5/13	6.9/14
MC	0.2/2	6/11	7.2/11
Bash	1/1	4/7	3/6
Pine	0.3/3	8/16	9.3/16
ProFTPd	0.9/2	21/63	7.8/18
Yafc	0.8/2	42/66	8.2/14
Transmission	1.2/5	64/172	12.4/18
W3M	0.7/4	134/199	8.7/15
MiniHTTP	1.4/2	46/88	9.2/14

ber of dependencies in process cache (column-3), and the average/maximum number of events cached in a system object (column-4). From the table, it is clear that the number of dependencies in the unit cache is quite small thanks to execution partitioning. The number of dependencies in the process caches varies for different applications. In most cases, the number of dependencies is less than 200. Firefox is the only one that reaches the size limit (256) and triggers the cache replacement. For the cache in each object, the average number is less than 10 events for most programs. Even for the maximum values, the average number is still less than 32 (the cache size).

To understand the behaviors of the caches, we ran Apache and Firefox, and counted the number of dependencies they cached over time. We set the process cache bound to a large number to observe the cache pressure. Figure 13 shows the results. For Apache, we used the ab benchmark [1] with 10 concurrent clients to generate the workload. For Apache, the number of dependencies varies in a small range and remains < 70. This is because each request has just a few read/write operations on the requested file and the socket (with the client), and cached dependencies are discarded when the corresponding files and sockets are closed. For Firefox, we performed two different experiments. The first one was a normal browsing. The blue line shows the result of this experiment. Firefox uses many system objects when it loads pages. After loading the page, many dependencies can be discarded as sockets are closed and temporary files are deleted. In our test scenario, the number of dependencies (in the process dependency cache) is around 150. The other experiment used a script to open a new web page in a new tab every second. The gray line shows the results. At the beginning, each new page caused a peak, and the script opened pages more frequently than the normal user. The number of dependencies is hence larger. After 10 minutes with 500+ pages opened, Firefox reached its capacity. The number of dependencies in the cache becomes flat. Even in this extreme situation, the number of dependencies is still reasonable due to the elimination of redundant and bogus dependencies.

Kernel Memory Consumption. Figure 14 shows the maximum kernel memory footprint caused by KCAL for each application. Since our cache sizes are fixed,

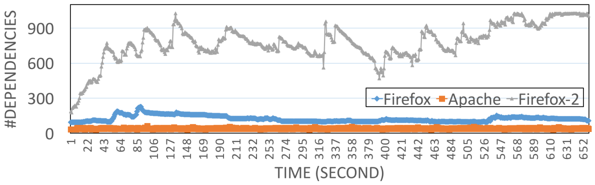


Figure 13: Number of dependencies in process dep. cache

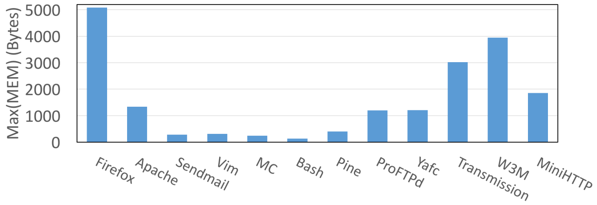


Figure 14: Max memory usage for individual applications

the memory overhead for each process including all its opened system resources (e.g., files/sockets) is fixed at 4224 bytes. In comparison, the original `task_struct` itself is 3520 bytes and it has a lot of pointers for opened system resources. One of its pointer fields `mm` pointing to a `mm_struct` is 952 bytes. Depending on the total number of system objects accessed to a process, the total memory footprint may vary a lot. However, since the number of events cached in an object tends to be small (Table 3), the kernel memory consumption is reasonably small, which ensures that KCAL does not perturb normal kernel functionalities.

Cache Size vs. Reduction Rate. The dependency cache sizes affect the reduction rate because evicting caches can result in keeping some redundant events. Previous experimental results indicate a small cache size is sufficient for many programs. In this experiment, we chose Firefox, whose dependency caches, especially the process cache, vary a lot over time, and we tested the effects of using different cache sizes. The results are presented in Figure 15. Even when the cache size is small, KCAL can still reduce many redundant events such as in-unit redundancies. With larger cache sizes, KCAL is able to detect and remove more cross-unit dependencies. If the cache is large enough (e.g., 1200 entries), all redundant dependencies are detected and the reduction rate is flat.

Comparison with State-of-the-Art ProTracer ProTracer can achieve a high reduction rate with a low runtime overhead (7% according to [24]). However, since ProTracer demands first generating the raw log before reduction, it requires parallel user-space processes to load and reduce the raw log. As a result, although its runtime overhead is low, the CPU consumption is substantial, because tainting on the large raw log files. Here we use the `ab` benchmark as an example to compare the CPU consumption of the two systems. Figure 16 shows the results.

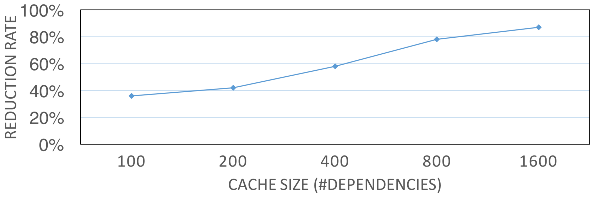


Figure 15: KCAL reduction results with different cache sizes

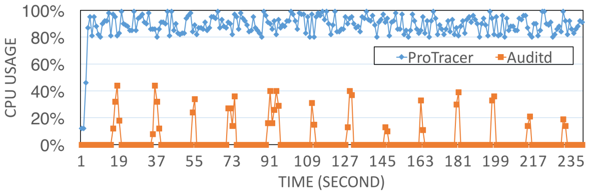


Figure 16: CPU consumption of ProTracer and KCAL

As seen in the graph, ProTracer uses the CPU consistently, and consumes almost all the cycles. In contrast, KCAL uses the CPU periodically. The average consumption is 4%. Even for the peaks, the CPU usage is about 40%, much less than ProTracer. This is because KCAL avoids generating the huge raw log in the first place and hence examines far fewer events for the same workload. In fact, ProTracer has to be pinned to a CPU to achieve the low runtime overhead. In contrast, KCAL's user-space processes just run as normal processes.

5 Conclusion

We analyzed the Linux Audit system and found the root cause of its high runtime and space overheads is its redundancy events. To solve this problem, we propose KCAL, a kernel-supported, cost-effective audit logging system for causality tracking. It caches dependencies and system events in the kernel and performs online log redundancy reduction. KCAL removes the overhead caused by transferring, processing, writing, and storing the redundant events. Our evaluation shows that KCAL can significantly reduce the log sizes and speed up the system.

6 Acknowledgements

We thank the anonymous reviewers for the valuable comments. In particular, we thank our shepherd, Howie Huang, for the constructive suggestions. This research was supported, in part, bythe United States Air Force and DARPA under contract FA8650-15-C-7562, NSF under awards1748764,1409668, and1320444, ONR under contracts N000141410468 and N000141712947, and Sandia National Lab under award1701331. Any opinions, findings and conclusionsor recommendations expressed in this materialare those of theauthors and do not necessarily reflect the views oftheUnited States Air Force and DARPA or othersponsors.

References

[1] Apache benchmark. https://httpd.apache.org/docs/2.2/programs/ab.html.

[2] Linux audit. https://people.redhat.com/sgrubb/audit/.

[3] Mozilla audit-go. https://github.com/mozilla/audit-go.

[4] Orchids. http://projects.lsv.ens-cachan.fr/orchidsdoc/.

[5] osquery. https://osquery.io/.

[6] Prelude siem. https://www.prelude-siem.org/.

[7] Sisdig. https://www.sysdig.org/.

[8] Bates, A., Butler, K. R., and Moyer, T. Take only what you need: Leveraging mandatory access control policy to reduce provenance storage costs. In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15)* (Edinburgh, Scotland, 2015), USENIX Association.

[9] Bates, A. M., Tian, D., Butler, K. R. B., and Moyer, T. Trustworthy whole-system provenance for the Linux kernel. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* (2015), J. Jung and T. Holz, Eds., USENIX Association, pp. 319–334.

[10] Borkin, M. A., Yeh, C. S., Boyd, M., Macko, P., Gajos, K. Z., Seltzer, M., and Pfister, H. Evaluation of filesystem provenance visualization tools. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (Dec. 2013), 2476–2485.

[11] Braun, U., Garfinkel, S. L., Holland, D. A., Muniswamy-Reddy, K., and Seltzer, M. I. Issues in automatic provenance collection. In *Provenance and Annotation of Data, International Provenance and Annotation Workshop, IPAW 2006, Chicago, IL, USA, May 3-5, 2006, Revised Selected Papers*, L. Moreau and I. T. Foster, Eds., vol. 4145 of *Lecture Notes in Computer Science*. Springer, 2006, pp. 171–183.

[12] Devecsery, D., Chow, M., Dou, X., Flinn, J., and Chen, P. M. Eidetic systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 525–540.

[13] Gehani, A., and Tariq, D. Spade: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference* (2012), Springer-Verlag New York, Inc., pp. 101–120.

[14] Goel, A., Po, K., Farhadi, K., Li, Z., and de Lara, E. The taser intrusion recovery system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2005), SOSP '05, ACM, pp. 163–176.

[15] Guo, Z., Zhou, D., Lin, H., Yang, M., Long, F., Deng, C., Liu, C., and Zhou, L. G2: A graph processing system for diagnosing distributed systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 27–27.

[16] Jakobsson, M., and Juels, A. Server-side detection of malware infection. In *Proceedings of the 2009 Workshop on New Security Paradigms Workshop* (New York, NY, USA, 2009), NSPW '09, ACM, pp. 11–22.

[17] Ji, Y., Lee, S., Downing, E., Wang, W., Fazzini, M., Kim, T., Orso, A., and Lee, W. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM, pp. 377–390.

[18] Kim, T., Wang, X., Zeldovich, N., and Kaashoek, M. F. Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 89–104.

[19] King, S. T., and Chen, P. M. Backtracking intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 223–236.

[20] King, S. T., Mao, Z. M., Lucchetti, D. G., and Chen, P. M. Enriching intrusion alerts through multi-host causality. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA* (2005), The Internet Society.

[21] Lee, K. H., Zhang, X., and Xu, D. High accuracy attack provenance via binary-based execution partition. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013* (2013), The Internet Society.

[22] Lee, K. H., Zhang, X., and Xu, D. Loggc: garbage collecting audit log. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), A. Sadeghi, V. D. Gligor, and M. Yung, Eds., ACM, pp. 1005–1016.

[23] Ma, S., Zhai, J., Wang, F., Lee, K. H., Zhang, X., and Xu, D. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association.

[24] Ma, S., Zhang, X., and Xu, D. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016* (2016), The Internet Society.

[25] Mehta, V., Bartzis, C., Zhu, H., Clarke, E., and Wing, J. Ranking Attack Graphs. *9th International Symposium on Recent Advances in Intrusion Detection (RAID'06) 4219* (2006), 127–144.

[26] Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Plale, B., Simmhan, Y., Stephan, E., and den Bussche, J. V. The open provenance model core specification (v1.1). *Future Gener. Comput. Syst. 27*, 6 (June 2011), 743–756.

[27] Muniswamy-Reddy, K.-K., Braun, U., Holland, D. A., Macko, P., Maclean, D., Margo, D., Seltzer, M., and Smogor, R. Layering in provenance systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 10–10.

[28] Muniswamy-Reddy, K.-K., Holland, D. A., Braun, U., and Seltzer, M. Provenance-aware storage systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), ATEC '06, USENIX Association, pp. 4–4.

[29] Ou, X., Boyer, W. F., and McQueen, M. a. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06* (2006), p. 336.

[30] Ou, X., Govindavajhala, S., and Appel, A. W. Mulval: A logic-based network security analyzer. 8–8.

[31] Pohly, D. J., McLaughlin, S., McDaniel, P., and Butler, K. Hi-fi: Collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 259–268.

[32] SAWILLA, R. E., AND OU, X. Identifying critical attack assets in dependency attack graphs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2008), vol. 5283 LNCS, pp. 18–34.

[33] SHEYNER, O., HAINES, J., JHA, S., LIPPMANN, R., AND WING, J. M. Automated generation and analysis of attack graphs. In *Proceedings - IEEE Symposium on Security and Privacy* (2002), vol. 2002-January, pp. 273–284.

[34] SITARAMAN, S., AND VENKATESAN, S. Forensic analysis of file system intrusions using improved backtracking. In *Third IEEE International Workshop on Information Assurance (IWIA'05)* (March 2005), pp. 154–163.

[35] SUNDARARAMAN, S., SIVATHANU, G., AND ZADOK, E. Selective versioning in a secure disk system. In *Proceedings of the 17th Conference on Security Symposium* (Berkeley, CA, USA, 2008), SS'08, USENIX Association, pp. 259–274.

[36] TIAN, D. J., BATES, A., BUTLER, K. R., AND RANGASWAMI, R. Provusb: Block-level provenance-based data protection for usb storage devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 242–253.

[37] VASUDEVAN, A., QU, N., AND PERRIG, A. Xtrec: Secure real-time execution trace recording on commodity platforms. In *Proceedings of the 2011 44th Hawaii International Conference on System Sciences* (Washington, DC, USA, 2011), HICSS '11, IEEE Computer Society, pp. 1–10.

[38] XIE, Y., FENG, D., TAN, Z., CHEN, L., MUNISWAMY-REDDY, K.-K., LI, Y., AND LONG, D. D. A hybrid approach for efficient provenance storage. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management* (New York, NY, USA, 2012), CIKM '12, ACM, pp. 1752–1756.

[39] XIE, Y., MUNISWAMY-REDDY, K., LONG, D. D. E., AMER, A., FENG, D., AND TAN, Z. Compressing provenance graphs. In *3rd Workshop on the Theory and Practice of Provenance, TaPP'11, Heraklion, Crete, Greece, June 20-21, 2011* (2011), P. Buneman and J. Freire, Eds., USENIX Association.

[40] XIE, Y., MUNISWAMY-REDDY, K.-K., FENG, D., LI, Y., AND LONG, D. D. E. Evaluation of a hybrid approach for efficient provenance storage. *Trans. Storage 9*, 4 (Nov. 2013), 14:1–14:29.

[41] XU, Z., WU, Z., LI, Z., JEE, K., RHEE, J., XIAO, X., XU, F., WANG, H., AND JIANG, G. High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 504–516.

[42] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 19–19.

[43] ZHU, N., AND CHIUEH, T.-C. Design, implementation, and evaluation of repairable file service. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* (June 2003), pp. 217–226.

EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs

Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, Binyu Zang

Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
{huazhichao123,dd_nirvana,xiayubin,haibochen,byzang}@sjtu.edu.cn

Abstract

The Meltdown vulnerability, which exploits the inherent out-of-order execution in common processors like x86, ARM and PowerPC, has shown to break the fundamental isolation boundary between user and kernel space. This has stimulated a non-trivial patch to modern OS to separate page tables for user space and kernel space, namely, KPTI (kernel page table isolation). While this patch stops kernel memory leakages from rouge user processes, it mandates users to patch their kernels (usually requiring a reboot), and is currently only available on the latest versions of OS kernels. Further, it also introduces non-trivial performance overhead due to page table switching during user/kernel crossings.

In this paper, we present EPTI, an alternative approach to defending against the Meltdown attack for unpatched VMs (virtual machines) in cloud, yet with better performance than KPTI. Specifically, instead of using two guest page tables, we use two EPTs (extended page tables) to isolate user space and kernel space, and unmap all the kernel space in user's EPT to achieve the same effort of KPTI. The switching of EPTs is done through a hardware-support feature called *EPT switching* within guest VMs without hypervisor involvement. Meanwhile, *EPT switching* does not flush TLB since each EPT has its own TLB, which further reduces the overhead. We have implemented our design and evaluated it on Intel Kaby Lake CPU with different versions of Linux kernel. The results show that EPTI only introduces up to 13% overhead, which is around 45% less than KPTI.

1 Introduction

The recently discovered Meltdown [16] and Spectre [14] vulnerabilities allow unauthorized processes to read data of privileged kernel or other processes, which brings severe security threat especially to cloud platforms. Currently, Intel has released micro-code patches to fix the Spectre vulnerability. However, in order to fix the Meltdown vulnerability, which is much more serious and easier to exploit, users are required to apply a kernel patch named KPTI (kernel page table isolation) [30] that uses two page tables to host kernel and user programs to isolate kernel address space from any user process. While this patch can effectively defend the Meltdown attacks, it brings three issues, which leaves thousands of millions of unpatched machines in danger.

First, the patch has to be applied manually by every user. In cloud environment, although the cloud administrators can patch the host OS, they cannot directly patch guest OS running in VMs (virtual machines) since they are not allowed to do so. For example, Amazon "recommend that customers patch their instance operating systems to address process-to-process or process-to-kernel concerns of this issue" [12]. However, many cloud users are not capable of doing such system maintenance.

Second, the patch may depend on specific versions of kernel, especially for Linux. Till now, Linux community just released version 4.15 that contains the patch. The patch may not work on some early versions of kernel like 4.4 [28]. It is expected to take a long time before the patch can be applied to all the versions of Linux kernel.

Third, the patch may incur non-trivial performance slowdown. The KPTI patch makes the kernel and user process use different page tables, which causes TLB-flush during the switching between user-mode and kernel-mode and thus increases the rate of TLB miss. Prior evaluation results show that for some system-call intensive workload, the performance penalty may be high as 30% in VMs [22]; our own experiments confirmed such performance slowdown (Section 6).

In this paper, we present an alternative approach to defending against Meltdown attack for VMs in cloud. Our approach, namely EPTI, can be applied to unpatched guest VMs without users' awareness and can achieve better performance than KPTI at the same time. First, instead of using two gPTs (guest page tables) as in KPTI, EPTI uses two EPTs (extended page tables), namely EPT_k and EPT_u, to run the kernel and user processes, correspondingly. The guest kernel and user still share one gPT, but in user mode, the gPT entries for mapping kernel address space are set to zero in EPT_u, which forbids any translation of address within kernel space to mitigate the Meltdown attack. Second, we leverage one of Intel's hardware features for virtualization, named *EPT switching*, to switch the two EPTs within the VM itself without causing any VMExit. We use binary instrumentation to insert two trampolines at the entrance and exit of guest kernel to do the EPT switching, which does not require kernel's source code and has little (if any) dependence on kernel versions. Third, through a detailed micro-architectural analysis, we find that EPT switching can be more efficient than gPT switching. Since each EPT has its own TLB, when switching the EPTs there will be no TLB flushing by hardware, which is the main reason of performance degradation of KPTI. We also adopt several optimizations to minimize the number of VMExits to further reduce the overhead. Fourth, EPTI can be seamlessly deployed in the cloud by combining with live VM migration [5]: a host can migrate away all the guest VMs, patch the host hypervisor with EPTI, and then migrate all the VMs back.

We have implemented EPTI on KVM and use unmodified Ubuntu distribution as guest VM for evaluation. We conduct a detailed security analysis as well as evaluation to show that our EPTI can achieve the same security guarantee as KPTI. We also evaluate real-world benchmarks to measure the performance overhead. The results show that the *average* performance overhead on server applications of EPTI is about 6%, which is 45% lower than KPTI whose *average* overhead is 11%.

To summarize, this paper makes the following contributions:

- An EPT-level isolation of kernel's and user's address spaces to defend against Meltdown attack for unpatched guest VMs.
- Several optimizations to achieve better performance than the current solution KPTI.
- A prototype of our design on real hardware for performance and security evaluation.

2 Motivation and Background

2.1 Meltdown Attack and KPTI

The Meltdown vulnerability was published in January 2018, known to affect Intel's x86 CPU, ARM Cortex-A75 [16] and some versions of PowerPC processor [11]. Through this attack, a malicious user application can steal contents of kernel memory in two steps. Step-1: to access kernel address *A* and to leverage its data as an index to access the cache; step-2: to get the data through cache covert channel. The key problem here is that the Step-1 is executed reordered and will be canceled eventually, but the cache layout is affected without rollback. Since the kernel will typically map all the physical memory within its memory space, the malicious application can potentially get all of the memory contents.

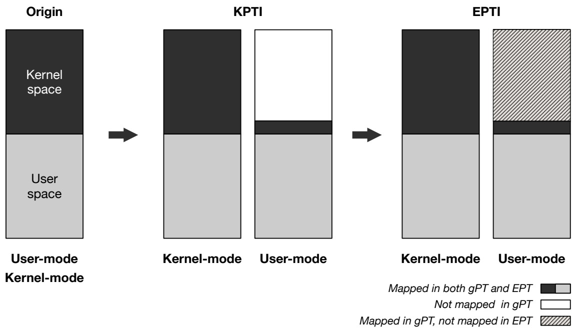


Figure 1: Page table isolation. For a VM, KPTI uses two gPTs and one EPT, while EPTI uses one gPT (since VM is not patched) and two EPTs.

KPTI (kernel page table isolation) [30] is based on KAISER (kernel address isolation to have side-channels efficiently removed) [19], which is proposed to defend against the Meltdown attack. This patch separates user space and kernel space page tables entirely, as shown in Figure 1. The one used by kernel is the same as before, while the one used by application contains a copy of user space and a small set of kernel space mapping with only trampoline code to enter the kernel. Since the data of kernel are no longer mapped in the user space, a malicious application cannot directly de-reference kernel's data address, and thus cannot issue Meltdown attack. KPTI has been merged to the mainstream Linux kernel 4.15, which was released on 28 Jan, 2018. However, the patch still has problems on previous Linux kernel versions. For example, it is reported that some Ubuntu user "just got the Meltdown update to kernel linux-image-4.4.0-108-generic but this does not boot at all" [28]. Considering the patch needs to be applied manually by system administrators, it may take a long time before most of the machines getting the patch deployed.

2.2 Overhead of KPTI

KPTI introduces performance overhead since both entering-kernel and exiting-kernel require additional page table switching. The switching is done by loading the CR3 register, which takes around 300 cycles. Meanwhile, since TLB (translation lookaside buffer) will be flushed during CR3 changing, the performance will further be affected due to higher TLB miss rate. There

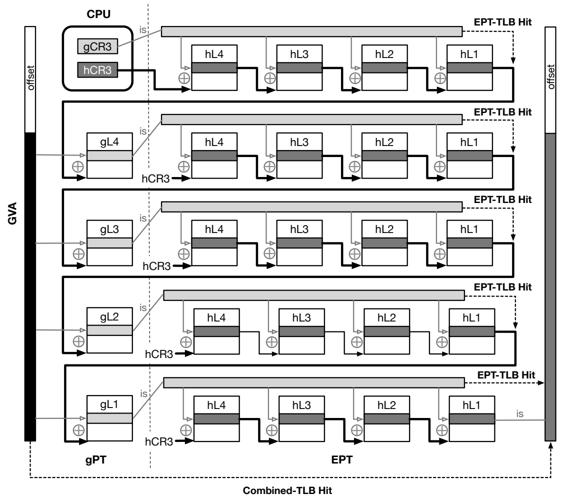


Figure 2: Process of translating GVA to HPA in an x86-64 guest VM. The gCR3 of CPU points to gPT and hCR3 points to EPT.

are many reports on evaluations of KPTI's overhead, which show that KPTI could lead to significant performance cost (up to 30%), particularly in syscall-heavy and interrupt-heavy workloads [30, 25, 18].

On processors that support PCID (process-context identifiers) feature, a TLB flush can be avoided and the performance overhead of KPTI can be reduced. PCID is a 12-bit tag of page table and is saved as a part of a TLB entry. For two page tables with different tags, their TLB entries can co-exist in the CPU and no TLB flush is needed when switching between the two page tables. Existing report shows that after enabling PCID, the overhead of KPTI on PostgreSQL's read-only test on Intel Skylake processor reduces from 17-23% to 7-17% [18]. However, Linux does not support PCID until version 4.14 released on 12 Nov 2017.

2.3 gPT, EPT and TLB

In native environment, PT (page table) is used for translating virtual address to physical address. In virtualization environment, the guest VM (virtual machine), running in non-root mode, only controls its GVA (guest virtual address) to GPA (guest physical address) mapping by gPT (guest page table). The hypervisor, running in root mode, controls each VM's GPA to HPA (host physical address) mapping through a hPT (host page table), which is called EPT (*extended page table*) on Intel platform [1].

Figure 2 shows the procedure of GVA-to-HPA translation on an x86-64 machine with 4 level gPT and EPT. The value of guest CR3 and addresses inside gPT are GPAs, while the value of EPTP and address inside EPT are HPAs. When CPU walks the gPT, it needs to translate all the GPA of needed gPT pages to HPA through EPT.

In order to minimize memory footprint during page walk, the processor has two types of TLB in virtualized environment: EPT-TLB and combined-TLB. The EPT-TLB is used for accelerating translation from GPA to HPA, while the combined-TLB stores entries of translation from GVA to HPA.

2.4 EPTP Switching with *VMFUNC*

VMFUNC is an Intel hardware virtualization extension, which provides VM functions for guest VMs, running in non-root mode, to directly invoke without VMExit. Currently, there is only one VM function provided by *VMFUNC*, named "EPTP switching", which allows software (either in the kernel mode or user mode) in guest VM to load a new EPTP (EPT pointer). Guest can only switch to the EPEP from a list of valid EPTP values configured by the hypervisor. The EPTP switching is supported on all Intel CPUs starting from Haswell architecture.

Performance of EPTP switching: We compare the latency of loading CR3 and EPTP switching. Writing the same value to the CR3 in guest VM costs around 300 cycles, with PCID enabled. While the "EPTP switching" takes about 160 cycles (two different EPTP values, but have the same mappings).

TLB behavior of EPTP switching: We further test the TLB behavior of EPTP switching and find how CPU constructs address mapping in TLB for different EPTs. The operations performed with one EPT will not affect other EPTs, Table 1 shows test results. In the table, "Invalid both EPTs' TLBs then fill EPT-0's TLB" means we first invoke *invlpg* instruction (which is used to flush TLB) in both EPT-0 and EPT-1, and then access the target memory in EPT-0. After that, we access the target memory again in both EPT-0 and EPT-1, and test the access latency. The result means that the EPT-0's is filled while the EPT-1's is not. We also test whether invoking flush TLB operations (write CR3 and *invlpg*) in one EPT will influence the other's TLB or not. We find that both of them flush other EPT's TLB.

3 System Overview

EPTI has three goals:

- **Goal-1**: To achieve the same security level as KPTI.
- **Goal-2**: To support protection on unpatched VMs seamlessly.
- **Goal-3**: To get better performance than KPTI.

[1] The hPT of AMD is called NPT (*nested page table*). Since the Meltdown attack only affects Intel's processor, we use "EPT" instead of "hPT" in the rest of the paper.

Table 1: TLB behaviors of different EPTs during *VMFUNC*.

Action	Access again in EPT-0	Access again in EPT-1	Conclusion
Invalid both EPTs' TLBs then fill EPT-0's TLB	3-5 cycles	120+ cycles	Each EPT has its own mapping in TLB.
Fill both EPTs' TLBs then write CR3 in EPT-0	120+ cycles	120+ cycles	Writing CR3 will flush TLB of all EPTs.
Fill both EPTs' TLBs then *invlpg* in EPT-0	120+ cycles	120+ cycles	*invlpg* will flush TLB of all EPTs.

We first construct two EPTs for each guest VM: EPT_k for kernel and EPT_u for user. The mapping of EPT_k is the same as original EPT (but with different permissions, which will be introduced later), so that the kernel will run just as before. When user applications are running, we need to ensure that they cannot access (even speculatively) any data in the kernel address space.

One intuitive way is to remove all the mappings of HPA of pages used by guest kernel in EPT_u, so that all kernel pages are not mapped when a user process is running. However, this solution does not work since typically Linux kernel will map *the entire GPA* to its GVA space, which is known as *direct mapping*, as shown in Figure 3. It means that we have to remove all of the GPA mappings from EPT, which will also disable the execution of user processes.

Instead, we just remap all the *gPT's pages that map kernel space* in EPT_u to a zeroed page, as shown in Figure 3. Thus, once a user process tries to access kernel address using its GVA, the GVA will never be translated to GPA since the CPU cannot find the corresponding mapping in gPT (refer to the left part of Figure 2). The security guarantee is the same as KPTI (**Goal-1**).

Next, we need to find a way to switch the EPTs at appropriate points. When a user process traps to kernel, the processor should immediately switch to EPT_k by *VMFUNC*. It also switches to EPT_u before the kernel returns to user process. In Linux kernel, there are limited entry points and exit points. The entry points can be located through IDT (interrupt descriptor table) and some specific MSRs (model-specific registers) [2]. The exit points must contain specific instructions (e.g., *sysretq*). Thus, we use *binary instrumentation* to re-write the kernel code on-the-fly to insert two pieces of trampoline code at the entry and exit points, which are mainly used to do the EPT switching. Leverage this method, EPTI can be used together with live migration to seamlessly protect a guest without rebooting it. More details are described in Section 4.3 (**Goal-2**).

In order to unmap the kernel space in EPT_u, we need to track which gPT pages are used for mapping kernel space, and zero them in EPT_u. EPTI tracks the gL3 pages, which are used to translate kernel GVA, and zero them (details in Section 4). We further present our optimizations in Section 5 to reduce the number of VMExits and get better performance (**Goal-3**).

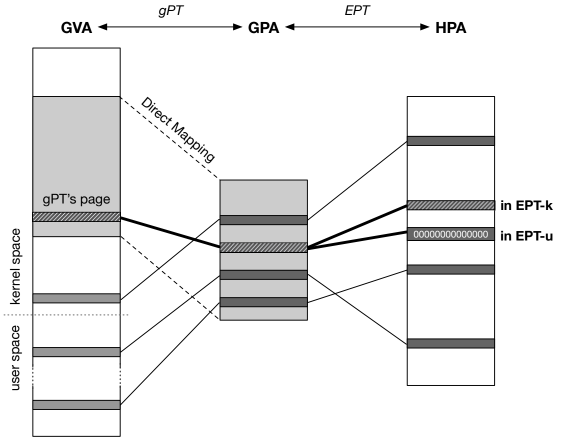


Figure 3: The difference of mapping of EPT_u and EPT_k.

Challenges: There are still many challenges on security and performance in the above design. For example, since it is allowed for a user process to invoke *VMFUNC*, a malicious process may try to switch to EPT_k before issuing Meltdown attack. We will describe our design with more details and present solutions to these challenges in the following text.

4 Design of EPT Isolation

In this section, we introduce the basic design of EPTI. Firstly, we need to construct the EPT_u, which removes all the kernel address mappings. Then, we introduce the basic method of how to track kernel gPT pages and add trampoline code for EPT switching. Finally we construct the EPT_k so that a malicious user cannot switch to it.

4.1 Zeroing gPT for Kernel Space in EPT_u

We remove all the GVA-to-GPA mappings of kernel address space in user mode by zeroing the gPT pages used for address translation in EPT_u. As shown in Figure 3, to zero a gPT page, we remap it to a new zeroed physical page in EPT_u. There are 4 page levels (from gL4 to gL1) in a 64-bit Linux kernel which uses 48-bit virtual address. Since each process has different gL4 pages , to minimize the modification to EPT_u, we only zero the gL3 pages used for kernel address translation (gPT structure is shown in Figure 5).

[2] E.g., IA32_LSTAR controls syscall entry.

After zeroing the gL3 pages for kernel space in EPT_u, accessing kernel memory from user mode will trigger a guest page fault since the target GVA is not mapped (although Meltdown attack can bypass permission check, it cannot access non-mapped pages). Since the kernel runs in EPT_k, it can never fill the zeroed gL3 page in EPT_u and the attacker's user process can never access the kernel memory (even speculatively).

4.2 Tracking gPT Pages in EPT_k

In order to zero all the gL3 pages that map kernel space in EPT_u, EPTI first needs to track all the gL3 pages for the kernel. Specifically, by setting the *CR3_LOAD_EXITING* bit in *VMCS* (virtual machine control structure), when a guest kernel changes CR3 it will trap to the hypervisor, which will then walk through the gPT to get a list of all gL3 pages for kernel space mapping. Meanwhile, the guest kernel may allocate new gL3 pages and add them to gL4. In order to track new kernel gL3 pages, all the gL4 pages will be mapped as read-only in EPT_k, so that each time a guest kernel adds a new gL3 page to gL4, it will trap to the hypervisor to update the monitored gL3 page list. We will present some optimization of tracking in Section 5.

4.3 Trampoline for EPT Switching

Listing 1 EPT switching to EPT_k

```
1:  SWITCH_EPT_K:
2:   SAVE_RAX_RCX
3:   movq $0, %rax
4:   movq $0, %rcx
5:   vmfunc
6:   RESTORE_RAX_RCX
```

The trampoline code contains instructions for EPT switching. Listing 1 shows the assembly code for switching from EPT_u to EPT_k. The *%rax* and *%rcx* contain the *VMFUNC index* and arguments passed to the *VMFUNC*. Line 3 means to call the first *VMFUNC* function (EPTP switching, index 0), and line 4 means to switch to EPT-0. Both *%rax* and *%rcx* are caller-saved, so the values need to be saved and restored in the trampoline. The process of *SWITCH_EPT_U* is similar but in the other direction.

Since the trampoline code is used to switch between EPT_k and EPT_u, it needs to be invoked in both EPTs. We need to ensure that: (1) the trampoline is executable in both EPTs and (2) there is a suitable place to store the caller-saved registers.

Mapping trampoline as executable in both EPTs: In EPT_u, only one page with the trampoline code will be mapped in the kernel space. To ensure that, EPTI remaps all the gPT pages (except gL4), which are used to translate the GVA for the trampoline, to new host physical pages. Then all the entries of these pages are set to zero, except those that used for mapping the trampoline (as shown in Figure 4). Entries of the guest IDT and the syscall entry MSR (IA32_LSTAR) will be changed to point to the trampoline code. In EPT_k, EPTI inserts the trampoline code to the end of direct map region of guest kernel, and re-writes the binary of kernel to change the exit points to *jmp* instructions that transfer control to the trampoline.

Saving caller-saved registers: Since *VMFUNC* will not save any register by hardware (which makes it fast), the trampoline code cannot use any register before saving them. One challenge to save these caller-saved registers is to support multi-core. For single CPU core, the value of %rax and %rcx can be saved to a memory page with a fixed address. However, for multi-core, one core may overwrite the saved register values of another core since they write to the same address.

Linux solves this problem by using *gs*-based per_cpu value. During system boot, it allocates a per_cpu memory region for each core. The base addresses of these regions will be recorded through *gs* registers of different cores after entering the kernel (through *swapgs* instruction) [3]. The following access of per_cpu values is performed by *%gs:index*. EPTI cannot leverage this method because: (1) it needs to know some specific semantics of the kernel and (2) it needs to map kernel's per_cpu region into EPT_u.

EPTI provides a per_vCPU memory page to save and restore the caller-saved registers. Specifically, a memory page is mapped into the kernel space in both EPT_u and EPT_k. To enable concurrent accesses from multiple cores, the page will be mapped to different physical pages for different vCPUs, so that when one vCPU saves %rax and %rcx, the values are written to its own page. In our implementation, we modify gPT to map this page to an unused GPA (e.g., the GPA out of the guest's DRAM range). In the EPTs for different vCPUs, we map this GPA to different HPA. In both EPT_k and EPT_u of one vCPU, it is mapped to the same HPA.

Seamless protection: Combined with live migration, EPTI can seamlessly protect a guest VM without rebooting it. The cloud provider can migrate away all the VMs, update the host hypervisor to enable EPTI and migrate all the VMs back. To resume a VM on EPTI, we need to: (1) map the trampoline into the guest; (2) rewrite the entries for interrupts and syscalls (stored in IDT and MSR), as well as the exit points (contain specific instructions e.g., sysretq), to jump to the trampoline; (3) enable the trapping of gPT and guest EPTP switching.

[3] The *swapgs* instruction exchanges the current *gs* value with the value stored in MSR_KERNELGSbase.

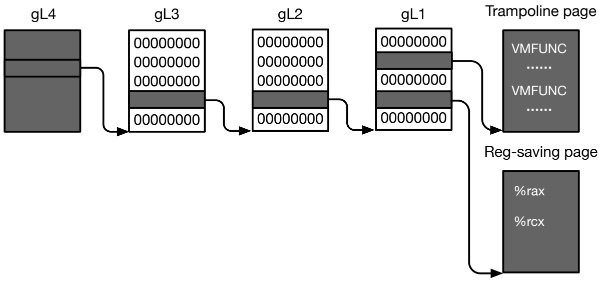


Figure 4: Contents of kernel space of EPT_u, which includes trampoline code page, register saving page, and gPT for mapping these two pages.

4.4 Malicious EPT Switching

The above design relies on an assumption that only the kernel can switch EPT. Unfortunately, the *VMFUNC* instruction can be invoked in either guest kernel mode or guest user mode, which enables an attacker to maliciously switch to EPT_k by *VMFUNC*, issue Meltdown attack and switch back to EPT_u. To defend this attack, EPTI needs to make EPT_k useless for the user process.

By performing real Meltdown attacks, we find that although Meltdown can read the memory without access permission, it cannot fetch code without executable permission even in reorder-execution. Base on this observation, we map all user memory as execute-never in EPT_k. Thus, once the user maliciously switches to EPT_k, all its code will be non-executable.

Specifically, EPTI only maps the guest physical memory (including kernel's code and kernel modules) as executable in EPT_k, and all other guest physical memory is mapped as execute-never. The kernel code is loaded during system booting and will not be changed during execution, EPTI can detect all the corresponding GPAs by searching gPT. The kernel modules are loaded/removed dynamically during runtime, EPTI needs to monitor all the guest physical pages used for them. This is done by trapping all write operations on gPT pages which translates GVA-to-GPA mapping of kernel modules. Since Linux kernel reserves a fixed GVA region for kernel modules, trapping modifications to the corresponding gPT pages will only influence the performance of installing/removing kernel modules.

5 Optimizations

As mentioned in the previous section, EPTI needs to trap both the load-CR3 operation and the write-gL4 in guest VMs. These trapping methods have three performance problems:

- **Useless traps of load-CR3**: EPTI traps guest VM's load-CR3 operations for getting all the gPTs. In fact, EPTI only needs to trap the *new* gPTs, but most of the load-CR3 operations just load old gPTs, which causes a lot of useless VMExits.
- **Access/Dirty bits update**: To trap the modification of a gPT page, EPTI marks it as read-only in EPT_k. However, for each memory access (including read, write and fetch), the CPU will update the A/D bits (access/dirty bits) in the gPT entries which are used for translating the target GVA, even when the A/D bits are already set by previous operations. Thus, whenever the kernel accesses any of its data, it will trigger an EPT violation, which causes a huge number of VMExit.
- **Additional traps of write-gPT**: In Section 4.2, EPTI traps all write-gL4 operations to track all enabled gL3 for kernel space mapping. However, each process has one gL4 page, which means EPTI needs to trap thousands of gL4 pages. Since kernel address mappings are the same for each process, trapping all these gL4 can be optimized.

In this section, we give several optimizations to solve all these performance problems.

5.1 Selectively Tracking Guest CR3

EPTI leverages a hardware feature to reduce the number of VMExit caused by trapping loading old CR3. Intel provides four *CR3_TARGET_VALUE* fields in *VMCS*. A load-CR3 in guest does not cause a VMExit if its source operand matches one of these values. We write the CR3 value, which 1) causes more than *threshold A* VMExits per second or 2) totally causes more than *threshold B* VMExits, to the *CR3_TARGET_VALUE* (A and B can be configured by the VMM).

5.2 Setting gPT Access/Dirty-Bit

In order to eliminate VMExit when CPU setting A/D-bit, we need to allow CPU to write gPT while disallowing kernel to do so. We find that the access path of them are different: the kernel writes gPT through its GVA (using both gPT and EPT), while the CPU writes gPT through its GPA (using EPT only). Thus, EPTI maps gPT pages with write permission in the EPT to allow CPU updating the A/D bits. To trap kernel modifying a gPT page, we redirect the GVA of this page to a new GPA which is mapped as read-only in EPT_k. This is done by (1) modifying the gL1 entry that is used for GVA-to-GPA translation of the target gPT page and remapping the gPT page to a new GPA; (2) mapping the new GPA to the original HPA as read-only, which contains the target gPT page. Thus, only the write access to the gPT page from kernel will trigger a VMExit.

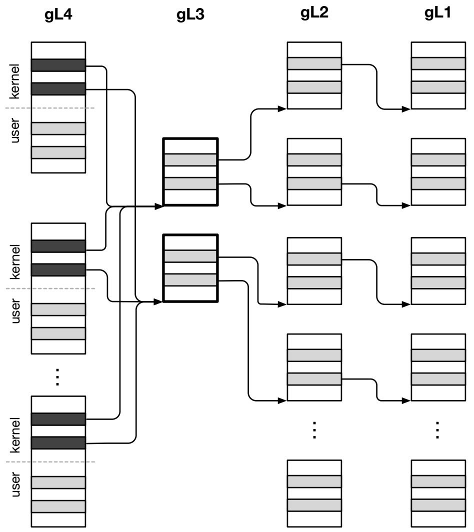


Figure 5: gPT of Linux. The kernel gL3 entries are shared by different gPTs.

5.3 Trapping gL3 Pages Instead of gL4

We adopt another optimization according to the following observations:

- *Most kernel virtual address regions are never changed.* Linux kernel reserve memory regions for different usages [4], and it never changes the mapping of most of these regions (e.g., direct map region is never changed).
- *Each gL3 pages can translate a large virtual space (512GB).* In a guest, it is almost impossible for the kernel to allocate so much virtual memory, so the number of kernel gL3 pages is rarely changed.
- In Linux kernel implementation, kernel only creates a new gL3 page when all entries of existing gL3 pages are in use, or the continuous free entries are not enough.

Based on the above observations, EPTI directly traps the modification of gL3 pages for kernel by default. When the last entry of a gL3 is used, which means the kernel may allocate a new gL3 page later, EPTI starts to trap the load-CR3 and write-gL4 until a new gL3 page is allocated. With this optimization, EPTI almost does not need to trap the operations of load-CR3 and write-gL4, which will reduce most (if not all) of VMExits.

6 Evaluation

In the evaluation, we try to answer these seven questions:

- *Question-1*: Can EPTI prevent Meltdown attacks?
- *Question-2*: How EPTI influences the performance of kernel critical operations (e.g., syscalls)?
- *Question-3*: How EPTI influences the performance of real server applications?
- *Question-4*: How EPTI influences the performance of multiple guest VMs?
- *Question-5*: How many VMExits are reduced by different optimizations of EPTI?
- *Question-6*: Can EPTI work on different kernel versions and how about the performance?
- *Question-7*: Can a guest VM be live migrated to hypervisor with EPTI and what is the performance?

6.1 Evaluation Environment

We do the evaluation on an x86-64 machine with an 8-core Intel Core i7-7700 CPU, 16GB memory and a Samsung 512GB SSD. We implemented EPTI with KVM based on Linux kernel 4.9.75 running in Ubuntu 14.04. We assigned 4 vCPUs (virtual CPUs) and 8GB memory to the guest VM, which runs an Ubuntu 16.04. The Linux kernel 4.9.75 is used as the guest kernel by default. In Section 6.4, we also test the overhead of multiple guest VMs. In Section 6.6, we test various kernel versions in the guest VM.

We isolate four physical cores on the host and each vCPU of the guest is pinned to a physical core. We use *virtio* to virtualize guest disk. During the evaluation, all the clients and server applications are running in the guest VM to reduce the influence of network.

In the performance evaluation, we test five systems: “Linux” (without KPTI), “KPTI” and EPTI with different optimizations, in which “EPTI-No” means EPTI with A/D bits updating, “EPTI-CR3” means applying A/D-bit updating and *CR3_TARGET_VALUE* to reduce VMExit caused by frequently loaded CR3, and “EPTI-CR3+L3” means applying all three optimizations.

6.2 Security Evaluation

First we implemented a PoC (proof of concept) of Meltdown attack, which has three steps: *step-1*: reads secret **S** from *kernel address*; *step-2*: uses **S** as an index to access memory (covert channel); and *step-3*: probes the cache and gets the value of **S**. The PoC also registers a signal handler of the segmentation fault to continuously perform the attack.

We use this PoC to steal *linux_proc_banner*, a value stored in kernel space (the PoC can also steal any other data in the kernel address space). It succeeds on Linux without KPTI, but is failed when using KPTI and EPTI. We then insert a *VMFUNC* in the PoC to make it switch to EPT_k just before step-1. The attack does not work

[4] e.g., In Linux with 48-bit VA, range from 0xffff880000000000 to 0xffffc7ffffffffff is used for direct map, and range from ffffc90000000000 to ffffe8ffffffffff is used for *vmalloc* and *ioremap*.

Table 2: Evaluation results of LMBench, in μs.

Operation	Linux	KPTI	EPTI-No	EPTI-CR3	EPTI-CR3+L3
Null syscall	0.04	0.16	0.12	0.12	0.12
Null I/O	0.07	0.2	0.17	0.17	0.16
Open/Close	0.70	0.93	0.84	0.84	0.83
Signal Handle	0.68	0.81	0.76	0.76	0.76
Fork syscall	72.9	79	80	80	75
Exec syscall	212	243	242	234	221
ctsw 16P/64K	6.07	7.37	7.66	7.66	6.39

on EPTI due to the defense mentioned in Section 4.4. We also try to pass a constant value through the covert channel after switching to EPT_k, which also fails.

The security evaluation shows that our system can successfully defend against existing Meltdown attacks, even if a malicious process switches to EPT_k first. Actually, a user process cannot execute any code in EPT_k. Logically, both EPTI and KPTI isolate the address space of user and kernel mode, so both of them can defend against Meltdown attacks.

6.3 Micro Benchmark

LMBench [21]: To answer *Question-2*, we use LMBench to test the time of some critical operations, e.g., syscall like *fork* and *exec*. As shown in Table 2, the *null* syscalls of unmodified Linux and KPTI take 0.04μs and 0.16μs, respectively. For EPTI, the time is about 0.12μs, which is smaller than KPTI due to the benefit of no-TLB-flushing of *VMFUNC*. The Null I/O, Open/Close and Signal Handle have the similar results as the *null* syscall. In all cases, EPTI performs better than KPTI. There is no difference between EPTI with different optimizations, because these operations do not involve load-CR3 or write-gL4.

For other three operations (*fork*, *exec* and context switch), EPTI-No and EPTI-CR3 take more time than EPTI-CR3+L3 because these operations contain many load-CR3 and write-gL4 operations. In LMBench, the EPTI-CR3 (optimized with *CR3_TARGET_VALUE*) has the same result with EPTI-No since a process is terminated before being identified as *trapping frequently*. The result of LMBench shows that both EPTI and KPTI have overhead on single critical operation, and the overhead of EPTI is smaller than KPTI.

SPEC_CPU 2006 [8]: We evaluate *all* of SPEC_CPU 2006 INT applications under five systems. As shown in Figure 6 (a), there is almost no overhead in both KPTI and EPTI, since these CPU-related applications rarely interact with the kernel.

6.4 Application Benchmark

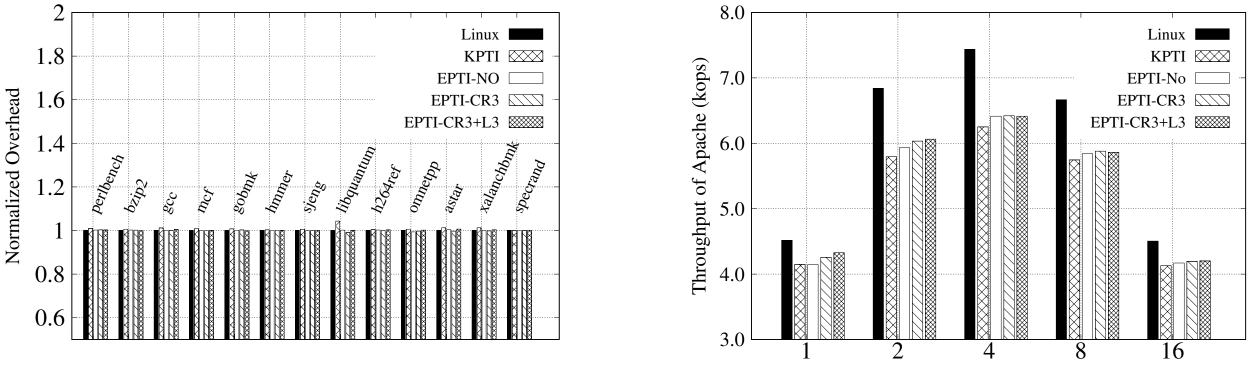
To answer that how EPTI influences the performance of server applications (*Question-3*), we evaluate the performance overhead of file system operations, databases and web servers.

Fs_mark [27] is used for evaluating file system performance. We configure it to continuously create 1MB files in the guest VM and use synchronization method 1 (call *fsync* before close a file), with different number of threads (each thread create 1000 files). The result is shown in Figure 7 (a), the KPTI has 6.5% overhead in single thread while our system has 1.1%. The overhead of both EPTI and KPTI are small because of the slow disk I/O performance for the guest.

Redis [24] is used for evaluating key-value store workloads. We use the standard redis-benchmark to test the throughput of *SET* and *GET* operation of Redis. The redis-benchmark is configured to use different numbers of threads (from 1 to 8) and the Redis runs with default configuration. Figure 7 (b) shows the result. The X-axis means the test operation and the number of threads used by the client (e.g., *SET-1* means *SET* operation with one thread). On the average, KPTI has about 12% of performance overhead while EPTI has 6%. In the worst case, KPTI has more than 20% overhead and our system has 12%.

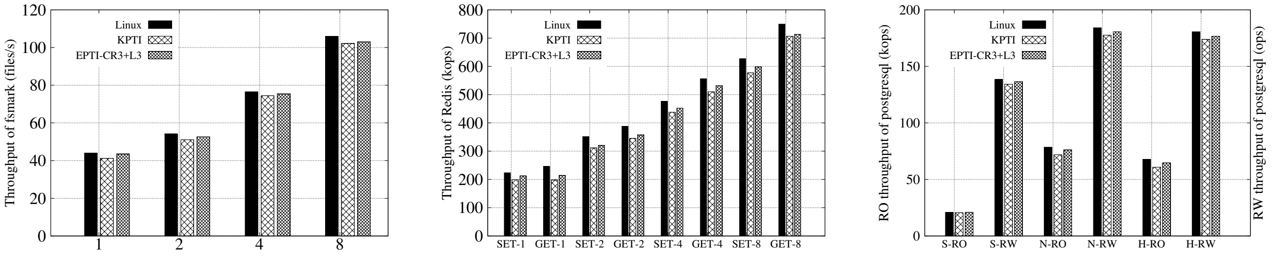
PostgreSQL [23] is a relational database. We test its performance with the *pgbench* (a benchmark provided by PostgreSQL based on TPC-B). We test the throughput of read-only and read-write transactions of PostgreSQL under three different loads: *single thread (S)*: using one database client; *normal (N)*: opening 4 test threads and 16 database clients; *heavy (H)*: opening 8 test threads and 64 database clients. The *pgbench* operates on a small database table with 1000 records. Each test is performed on a cleaned table and lasts for 60 seconds. The PostgreSQL is running with default configuration. The result is shown in Figure 7 (c). The unit of throughput of RO transaction is *kops* and the unit of RW is *ops*. Both KPTI and EPTI have small overhead for single thread *pgbench*. The overhead increases dramatically in the normal and heavy loads. In the Heavy-RO test, KPTI has about 12% overhead while our system has about 4%.

MongoDB [3] is a widely-used non-relational database. We use YSCB benchmark to test the performance of it with different workloads. Each workload is performed on a table with 10,000 records and we configure YCSB to use 32 test threads. The MongoDB uses the default configuration. We test all the standard workloads of YSCB (from workload-A to workload-F) and the result is shown in Figure 8 (a). On average, KPTI has about 7% performance overhead while our system has about 2%.



(a) SPEC_CPU INT **(b) Apache**

Figure 6: Figure (a) shows the overhead of all INT applications in SPEC_CPU 2006 benchmark, lower the better. Figure (b) shows the throughput of Apache with different number of clients, higher the better.



(a) fs_mark **(b) Redis** **(c) PostgreSQL**

Figure 7: Figure (a) shows the throughput of fs_mark with different threads. Figure (b) shows the throughput of SET and GET operations of Redis with different threads used by the client. Figure (c) shows the throughput of PostgreSQL under different workloads of RO (read-only) and RW (read-write) transactions. Higher the better.

Apache [1] is a widely-used web server. We use *ab* (apache benchmark) to test the throughput of Apache. It continuously downloads a 1MB static web page from the Apache, with different client threads (1 to 16). The Apache server uses default configuration (event mode). Figure 6 (b) shows the throughput of Apache. The performance drops after 4 client threads since we use 4 vCPUs in the VM. The overhead of KPTI is 15%-18%, while our system (EPTI-CR3+L3) has about 10% overhead.

Nginx [2] is a lightweight web server. We also test it by *ab* benchmark with a 1MB static web page and different threads (1 to 16). The Nginx server runs with default configuration. The throughput of Nginx is shown in Figure 8 (b). In the worst case, the overhead of KPTI is 18% and ours is 12%.

Multiple guest VMs: We evaluate the overhead of EPTI on multiple guest VMs (for *Question-4*). Each VM is configured to have 1 vCPU and 1GB memory, and all the VMs' vCPUs are pinned on 4 physical cores. We use linux 4.15 as the guest kernel, and run a Nginx server as well as an *ab* benchmark tool in each VM. The result is shown in Figure 9, the average overhead of KPTI is about 16% while our system is about 9%

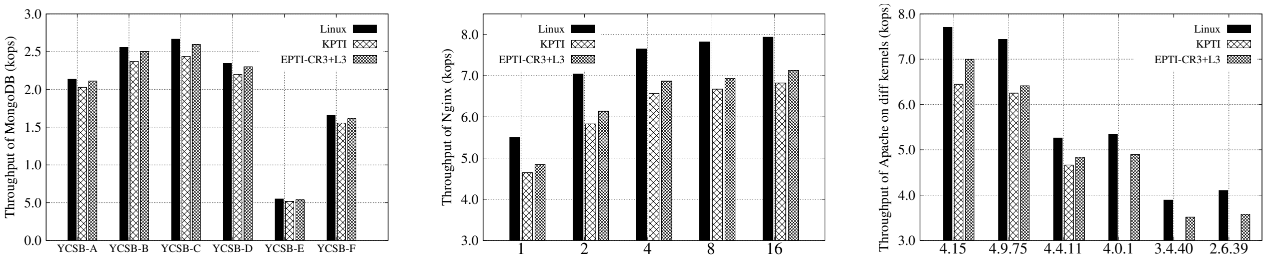
Table 3: Number of VMExit caused by EPTI.

Benchmark	EPTI-No	EPTI-CR3	EPTI-CR3+L3
Redis 1-thread	540	464	0
Redis 8-thread	385	315	0
Apache 4-thread	45406	225	0
Apache 32-thread	40149	623	0
Compile Kernel -j8	609659	551023	0

6.5 Breakdown of Optimizations

To answer how different optimizations reduce the number of VMExit of EPTI (*Question-5*), we test the performance of Apache on EPTI with different optimizations. Figure 6 (b) shows the result. In the best case (1 client thread), EPTI-No has about 9% performance overhead which is almost same as KPTI. EPTI-CR3 only has 5% overhead while EPTI-CR3+L3 has 4%.

To give a detailed breakdown of the performance improvement, we analyze the number of VMExits in EPTI with different optimizations. We calculate the total number of VMExits caused by EPTI of the whole guest in three scenarios: (1) running redis-benchmark to test Redis (1,000,000 operations with 1 or 8 threads); (2) running ab to download 1,000 1MB web pages (with 4 or



(a) MongoDB **(b) Nginx** **(c) Apache on different kernel versions**

Figure 8: Figure (a) and (b) show the throughput of MongoDB and Nginx, higher the better. For MongoDB, we test it with YSCB and X-axis means the different YSCB workloads. For Nginx, we test it by using ab benchmark with different threads. Figure (c) shows the throughput of Apache on different kernel versions, X-axis means the kernel version.

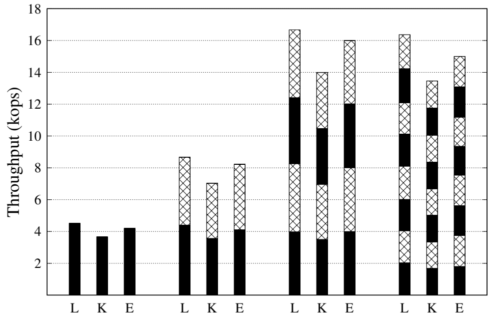


Figure 9: Throughput of Nginx on multiple guest VMs. (L means Linux, K means KPTI, E means EPTI-CR3+L3)

32 client threads); and (3) compiling Linux kernel 4.9.75 with "defconfig" (including kernel modules, "make -j8"). The result is shown in Table 3.

As shown in the table, the optimization of selectively trapping load-CR3 does not have much effect on Redis and kernel compilation, but is effective for Apache. This is because EPTI-CR3 can only reduce the VMExit caused by frequently loading CR3 value, while both Redis and redis-benchmark are single-process that have very few load-CR3 or write-gL4 operations. In kernel compilation, the Makefile creates a gcc process to compile each C file, which produces a huge number of processes with different CR3. Since each gcc process works for a short time, there is no long-term frequently-used CR3 which means the EPTI-CR3 cannot effectively reduce the number of VMExits (theoretically, the result of EPTI-CR3 can be improved by a better algorithm for replacing the value of *CR3_TARGET_VALUE*). On the contrary, Apache with event mode uses a few (typically 4) child processes to manage all the worker threads. Most of the VMExits are caused by loading the CR3 of Apache's child process, which can be optimized by storing their CR3 in *CR3_TARGET_VALUE*.

For all scenarios, there is no modification on kernel gL3, so the number of VMExit can be further reduced to 0 by EPTI-CR3+L3. The reason we still need both optimizations is that operation on kernel gL3 is highly OS dependent, while the optimization of selectively trapping CR3 is more general.

Table 4: VM live migration to host with EPTI, in *ms*.

	KVM w/o EPTI	KVM w/ EPTI
Total time	15779.5 ±1112.03	15782.6 ±1111.86
Downtime	6.1 ±0.82	9.2 ±1.03

6.6 Different Kernel Versions

To answer *Question-6* (could EPTI works on different kernel versions and how about their performance?), we test the performance of EPTI on different Linux kernel versions (selected from 2.6 to 4.15). We run Apache on them and use *ab* benchmark with 4 client threads to evaluate the throughput. The result is shown in Figure 8 (c). Our system has higher performance than KPTI in all the kernel versions (excluding kernels which do not have KPTI support). In the newest kernel 4.15, which enabled PCID, the performance of Linux w/o KPTI is improved obviously. However the KPTI of Linux 4.15 still has about 17% overhead, while our system has 10%.

6.7 VM Migration

To answer the last question, we test the total time and downtime of VM live migration, from a host without EPTI to one with EPTI. The source machine deploys an unmodified Linux kernel 4.9.75 with the same hardware configuration as mentioned, and the target is the one we use in previous evaluation. We use both the unmodified KVM and KVM with EPTI as the target hypervisor. A guest VM can be seamlessly migrated to a hypervisor with EPTI and the overhead is small. Table 4 shows the average migration time together with the stddev (test for 4 times). The downtime increases 3 ms which is caused by the scanning of code region in memory, preparing for two EPTs and binary writing.

7 Related Work

Besides the work mentioned, we now present the systems that also leverage similar hardware features for enhancing system security or performance.

KAISER [19] was proposed to defend against attacks on KASLR [10, 7, 13], which can also prevent Meltdown since it ensures no valid mapping to kernel space in user mode. It is later replaced by KPTI [30] and is merged to Linux kernel from version 4.15.

SecVisor [26] ensures lifetime kernel integrity via setting access permissions in NPT (Nested Page Table, from AMD, similar to Intel's EPT). TrustVisor [20] uses NPT to isolate memory regions used by a security task. CloudVisor [31] de-privileges the hypervisor to non-root mode and uses a separated EPT to host it. Thus, the hypervisor is isolated from guest VMs and is removed from the TCB (trusted computing base). InkTag [9] uses EPT to isolate the address space of a process. SeCage [17] leverages *VMFUNC* to provide two isolated execution environments, one for running security-critical code and the other for the normal code, to defend against attacks like heartbleed [29]. Similarly, MemSentry [15] creates domains (VMs) to hide secret data and uses *VMFUNC* to switch between different domains.

8 Discussion

Supporting x86-32: To support 32-bit linux, EPTI needs two steps, 1) trapping and modifying the gPT and 2) inserting the trampoline. The existing design can be used to trap and construct gPT for 32-bit linux. To add the trampoline, EPTI requires 8KB virtual address region within guest VM which should not be occupied by the VM itself. We could use technology like *shadow IDT* of ELI [6], which leverages extra pages of devices PCI BAR (base address register) in guest VM to insert additional pages.

Supporting five-level page table: 64-bit Linux also provides five-level page table (the root gPT is gL5). EPTI can trap all enabled gL4 pages and zero them in EPT_u to perform the user-kernel isolation. All the trap and zero methods are same as the four-level page table.

Supporting Windows: Technically, the design of EPTI can be applied to Windows or other OSes. All the kernel entries of Windows kernel can be detected by trapping the modification of IDT and MSRs, so that a trampoline can further be added. After that, EPTI can construct the EPT_u and EPT_k after knowing the virtual memory layout of Windows kernel.

Transparency to guest VMs: EPTI modifies the code and gPT of the guest. In current implementation, these modifications are not transparent to the guest VM. These modifications will not affect functionalities like VMI (virtual machine introspection) and kernel integrity check. For the VMI case, we keep the original address mapping with only different permission so the address translation in VMI can be done as before. For the kernel integrity check case, we do not change existing kernel code, so that its hash value will not be changed. Moreover, the VMM can make the modifications transparent to the guest. Features like *XnR* (execute-no-read) [4] can be used to prevent kernel from reading the trampoline code page while still allowing to execute the code, and the access to the modified gPT pages can also be trapped.

Compared with KPTI: EPTI has three advantages compared with KPTI: compatibility, performance and seamless deployment. Even when the KPTI is patched on all Linux versions, EPTI is still valuable for its low performance overhead and seamless deployment without guest rebooting.

9 Conclusion

The publish of Meltdown vulnerability makes public servers in danger, especially those in cloud. The KPTI solution requires server owners to apply the patch manually, which currently supports only a few of kernel versions and may introduce non-negligible performance overhead. This paper presents EPTI, a new solution to Meltdown vulnerability that can be applied to unpatched VMs and with less overhead. Specifically, our solution uses two EPTs (extended page tables) to isolate user space and kernel space, and unmaps all the kernel space in user's EPT to achieve the same effort of KPTI. EPTI leverages two hardware features to reduce the performance overhead: first, the switching of EPTs is done through a hardware-support feature called *EPTP switching* within guest VMs without hypervisor involvement. Second, *EPTP switching* does not flush TLB since each EPT has its own TLB, which further reduces the overhead. By leveraging live migration, EPTI can seamlessly protect a guest VM without rebooting it. We have implemented our design and evaluated on Intel Kaby Lake CPU with different versions of Linux kernel. The results show that EPTI only introduces up to 13% overhead, which is around 45% less than KPTI.

Acknowledgments

This work was supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000104, the National Natural Science Foundation of China under Grant Nos. 61572314 and 61525204.

References

[1] Apache http server. `https://www.apache.org/`. Referenced Feb 2018.

[2] Apache http server. `https://nginx.org/cn/`. Referenced Feb 2018.

[3] Mongodb. `https://www.mongodb.com/`. Referenced Feb 2018.

[4] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1342–1353.

[5] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2* (2005), USENIX Association, pp. 273–286.

[6] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. Eli: bare-metal performance for i/o virtualization. *ACM SIGPLAN Notices 47*, 4 (2012), 411–422.

[7] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (2016), ACM, pp. 368–379.

[8] HENNING, J. L. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News 34*, 4 (2006), 1–17.

[9] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. Inktag: Secure applications on an untrusted operating system. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 265–278.

[10] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 191–205.

[11] IBM. Potential impact on processors in the power family. `https://www.ibm.com/blogs/psirt/potential-impact-processors-power-family/`. Referenced Jan 2018.

[12] INC., A. Processor speculative execution research disclosure. `https://aws.amazon.com/cn/security/security-bulletins/AWS-2018-013/`. Referenced Jan 2018.

[13] JANG, Y., LEE, S., AND KIM, T. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 380–392.

[14] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints* (Jan. 2018).

[15] KONING, K., CHEN, X., BOS, H., GIUFFRIDA, C., AND ATHANASOPOULOS, E. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 437–452.

[16] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *ArXiv e-prints* (Jan. 2018).

[17] LIU, Y., ZHOU, T., CHEN, K., CHEN, H., AND XIA, Y. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1607–1619.

[18] MAILING LIST, P. heads up: Fix for intel hardware bug will lead to performance regressions. `https://www.postgresql.org/message-id/20180102222354.qikjmf7dvnjgbkxe@alap3.anarazel.de`. Referenced Jan 2018.

[19] MAURICE, C., AND MANGARD, S. Kaslr is dead: Long live kaslr. In *Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings* (2017), vol. 10379, Springer, p. 161.

[20] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 143–158.

[21] MCVOY, L. W., STAELIN, C., ET AL. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference* (1996), San Diego, CA, USA, pp. 279–294.

[22] NEWS, H. Kpti overhead of redis. `https://news.ycombinator.com/item?id=16079457`. Referenced Jan 2018.

[23] POSTGRESQL. Postgresql database. `https://www.postgresql.org`. Referenced Feb 2018.

[24] REDIS. Redis database. `https://redis.io`. Referenced Feb 2018.

[25] REGISTER, T. Kernel-memory-leaking intel processor design flaw forces linux, windows redesign. `https://www.theregister.co.uk/2018/01/02/intel_cpu_design_flaw/`. Referenced Jan 2018.

[26] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 335–350.

[27] SOURCEFORGE. Fs_mark. `https://sourceforge.net/p/fsmark/wiki/Home/`. Referenced Feb 2018.

[28] UBUNTU. Meltdown update kernel does not boot. `https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1742323`. Referenced Jan 2018.

[29] WIKIPEDIA. Heartbleed. `https://en.wikipedia.org/wiki/Heartbleed`. Referenced Jan 2018.

[30] WIKIPEDIA. Kernel page-table isolation. `https://en.wikipedia.org/wiki/Kernel_page-table_isolation`. Referenced Jan 2018.

[31] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 203–216.

Effectively Mitigating I/O Inactivity in vCPU Scheduling

Weiwei Jia[12]*, Cheng Wang*[1]*, Xusheng Chen*[1]*, Jianchen Shan*[2]*, Xiaowei Shang*[2]
Heming Cui[1]*, Xiaoning Ding*[2]*, Luwei Cheng*[3]*, Francis C. M. Lau*[1]*, Yuexuan Wang*[1]*, Yuangang Wang*[4]
[1] *University of HongKong* [2] *New Jersey Institute of Technology* [3] *Facebook* [4] *Huawei*

Abstract

In clouds where CPU cores are time-shared by virtual CPUs (vCPU), vCPUs are scheduled and descheduled by the virtual machine monitor (VMM) periodically. In each virtual machine (VM), when its vCPUs running I/O bound tasks are descheduled, no I/O requests can be made until the vCPUs are rescheduled. These inactivity periods of I/O tasks cause severe performance issues, one of them being the utilization of I/O resources in the guest OS tends to be low during I/O inactivity periods. Worse, the I/O scheduler in the host OS could suffer from low performance because the I/O scheduler assumes that I/O tasks make I/O requests constantly. Fairness among the VMs within a host can also be at stake. Existing works typically would adjust the time slices of vCPUs running I/O tasks, but vCPUs are still descheduled frequently and cause I/O inactivity.

Our idea is that since each VM often has active vCPUs, we can migrate I/O tasks to active vCPUs, thus mitigating the I/O inactivity periods and maintaining the fairness. We present vMIGRATER, which runs in the user level of each VM. It incorporates new mechanisms to efficiently monitor active vCPUs and to accurately detect I/O bound tasks. Evaluation on diverse real-world applications shows that vMIGRATER can improve I/O performance by up to 4.42X compared with default Linux KVM. vMIGRATER can also improve I/O performance by 1.84X to 3.64X compared with two related systems.

1 Introduction

To ease management and save energy in clouds, multiple VMs are often consolidated on a physical host. In each VM, multiple vCPUs often time-share a physical CPU core (aka. pCPU). The VMM controls the sharing by scheduling and descheduling the vCPUs periodically. When a vCPU is scheduled, tasks running on it become active and make progress. When a vCPU depletes its time slice, it is descheduled, and tasks on it become inactive and stop making progress.

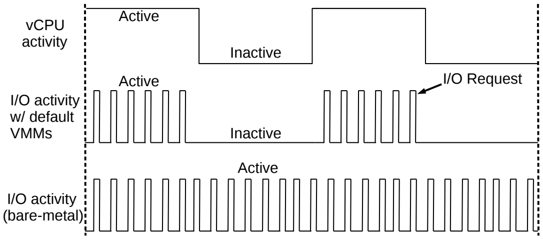


Figure 1: ***I/O inactivity*.**

vCPU inactivity leads to a severe I/O inactivity problem. After the vCPU is descheduled, the I/O tasks on it become inactive and cannot generate I/O requests, as shown in the first two curves in Figure 1. The inactive periods can be much longer than the latencies of storage devices. Typical time slices can be tens of milliseconds; the storage device latencies are a few milliseconds for HDDs and microseconds for SSDs. Thus, during the I/O inactive periods, I/O devices (both physical and virtual devices) may be under-utilized. The under-utilization becomes more serious with a higher consolidation rate (i.e., the number of vCPUs shared on each pCPU), because a vCPU may need to wait for multiple time slices before being rescheduled. The I/O throughput of a VM drops significantly with a consolidation rate of 8 as recommended by VMware [47], based on our evaluation in Section 5.

The I/O inactivity problem becomes even more pronounced when I/O requests are supposed to be processed by fast storage devices (e.g., SSDs). Usually, a vCPU remains active during I/O requests, so it can quickly process them. A similar situation is when a computation task and an I/O task run on the same vCPU. When the I/O task issues a read request and then waits for the request to be satisfied, the computation task is switched on. At this moment, the vCPU is still running; thus, when the read request is satisfied, the vCPU can quickly respond to the event and switch the I/O task back. However, if the time slice of the vCPU is used up by the computation task in one scheduling period, the I/O task cannot proceed until the next period, causing the I/O task to be slowed down significantly.

Worse, the I/O inactivity problem causes the I/O scheduler running in the host OS to work extremely ineffectively. To fully utilize the storage devices, based on the latencies of I/O devices, system designs would carefully control the factors affecting the latencies experienced by I/O workloads (e.g., wake-up latencies and priorities). Thus, I/O workloads running on bare-metal can issue the next request after the previous request is finished. I/O inactive periods make these mechanisms ineffective. Moreover, non-work-conserving I/O schedulers [40] would often hold an I/O request until the next request from the same I/O task comes in (refer to §2.2). By serving the requests from the same task continuously, which have better locality than requests from different tasks, such I/O schedulers [40] can improve I/O throughputs. However, since an I/O workload cannot continue to issue I/O requests after its vCPU becomes inactive, the I/O scheduler in the host OS must switch to serve the requests from other I/O tasks, which greatly reduces locality and I/O throughput, as we will show in our evaluation.

Last but not least, the I/O throughput of a VM can be "capped" by its amount of CPU resources. If the vCPUs in a VM (VM_a) are assigned with smaller proportions of CPU time on each pCPU than the vCPUs on another VM (VM_b), the I/O workloads on VM_a will get less time to issue I/O requests and may only be able to occupy a smaller proportion of the available I/O bandwidth. Since the actual I/O throughputs of the VMs are affected by both I/O scheduling and vCPU scheduling, it is difficult for the I/O scheduler to ensure fairness between the VMs.

All the above problems share the same root cause, I/O inactivity, and existing works mainly try to curb vCPU inactivity but ignore this root cause. Existing works primarily follow two approaches: 1) shortening vCPU time slices (vSlicer [49]); and 2) assigning higher priority to I/O tasks running on active vCPUs (xBalloon [44]). Unfortunately, vCPUs with either approach are still descheduled frequently and cause I/O inactivity.

Since a VM often has active vCPUs, our idea to mitigate I/O inactivity is to try to efficiently migrate I/O tasks to active vCPUs. By evenly redistributing I/O tasks to active vCPUs in a VM, I/O inactivity can be greatly mitigated and I/O tasks can make progress constantly. This maintains both performance and fairness for I/O tasks as they are running on bare-metal. The fairness of I/O bandwidth among VMs on the same host is also maintained.

We implement our idea in vMIGRATER, a user level tool working in each VM. It is transparent as it does not need to modify application, OS in VM, or VMM. vMIGRATER carries simple and efficient mechanisms to predict whether a vCPU will be descheduled and to migrate the I/O tasks on this vCPU to another active vCPU.

vMIGRATER adds only small overhead to applications for two reasons. First, I/O bound tasks use little CPU time, so the I/O tasks migrated by vMIGRATER hardly affect the co-running tasks on the active vCPUs. Second, vMIGRATER migrates more I/O bound tasks to the active vCPUs with more remaining time slices, so all vCPUs' loads in the same VM are well balanced. By reducing I/O inactivity with low overhead, vMIGRATER makes applications run in a fashion similar to what they do on bare-metal, as shown in Figure 1.

vMIGRATER has to address three practical issues. First, it needs to identify I/O tasks. To address this issue, vMIGRATER uses an event-driven model to collect I/O statistics and to detect I/O bound tasks quickly. Second, vMIGRATER needs to determine when an I/O bound task should be migrated. To minimize overhead, vMIGRATER only migrates an I/O bound task when the vCPU running this task is about to be descheduled. vMIGRATER monitors each vCPU's time slice and uses the length of the previous time slice to predict the length of the current time slice. Third, vMIGRATER needs to decide where a task should be migrated to keep it active. Based on the collected time slice and I/O task information, vMIGRATER migrates I/O tasks from to-be-descheduled vCPUs to the active vCPUs with light workload.

We implemented vMIGRATER in Linux and evaluated it on KVM [30] with a collection of microbenchmarks and 7 widely used or studied programs, including small programs (sequential, random and bursty read) from SysBench [7], a distributed file system HDFS [5], a distributed database Hbase [2], a mail server benchmark PostMark [6], a database management system LevelDB [3], and a document-oriented database program MongoDB [35]. Our evaluation shows that:

1. vMIGRATER can effectively improve application throughput. Compared to vanilla KVM, vMIGRATER can improve application throughputs by up to 4.42X. With vMIGRATER, application throughput is 1.84X to 3.64X higher than vSlicer and xBalloon.
2. The effectiveness of vMIGRATER increases with consolidation rate. Compared to vanilla KVM, vMIGRATER improves application throughput from 1.72X to 4.42X when the number of consolidated VMs increases from 2 to 8.
3. vMIGRATER can maintain the fairness of the I/O Scheduler in VMM. Compared to vanilla KVM, vMIGRATER reduces unfairness between VMs by 6.22X. When VMs are assigned with the same I/O priority but different CPU time shares, the VMs can still utilize similar I/O bandwidth.

The paper makes the following contributions. First, the paper identifies I/O inactivity as a major factor degrading I/O throughputs in VMs, and quantifies the severity of the problem. Second, it designs vMIGRATER, a simple and practical user-level solution, which greatly improves the throughput of I/O applications in VMs. Third, vMIGRATER is implemented in Linux, and is evaluated extensively to demonstrate its effectiveness.

The remainder of this paper is organized as follows. §2 introduces the background and motivation of vMIGRATER. §3 presents the design principles, architecture, and other design details of vMIGRATER. §4 describes implementation details. §5 presents evaluation results. §6 introduces related work, and §7 concludes the paper.

2 Background and Motivation

This section first introduces vCPU scheduling (§2.1) and I/O request scheduling (§2.2) as the background. Then it explains three performance problems caused by I/O inactivity in virtualized systems (§2.3) to motivate our research.

2.1 vCPU Scheduling

To improve resource utilization in virtualized systems, a pCPU is usually time-shared by multiple vCPUs. A vCPU scheduler is used to periodically deschedule a vCPU and schedule another vCPU. For instance, KVM uses completely fair scheduler (CFS) [13, 44] to schedule vCPUs onto pCPUs. CFS uses virtual runtime (vruntime) to keep track of the CPU time used by each vCPU and to make scheduling decisions. With a red-black tree, it sorts vCPUs based on their vruntime values, and periodically schedules the vCPU with the smallest vruntime value. In this way, CFS distributes time slices to vCPUs in a fair way.

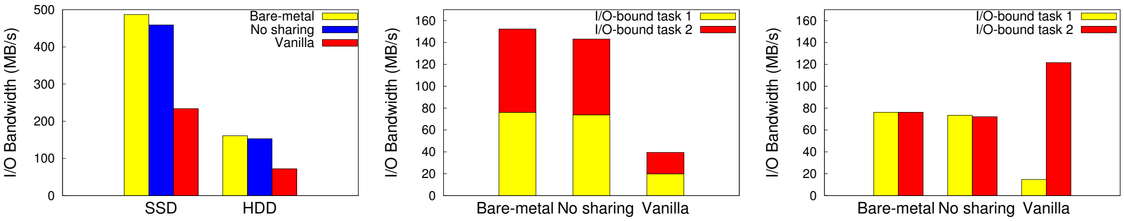
2.2 I/O Request Scheduling

I/O requests are scheduled by the I/O scheduler in the VMM. There are two types of I/O schedulers: work-conserving schedulers [19, 38] and non-work-conserving schedulers [51, 27]. A work-conserving I/O scheduler always keeps the I/O device busy by scheduling pending I/O requests as soon as possible.

Non-work-conserving I/O schedulers, such as anticipatory scheduler (AS) [27] and Completely Fair Queuing (CFQ) [12], are now widely used. A non-work-conserving scheduler waits for a short period after scheduling a request from a task, expecting that other requests from the same task may arrive. Because requests from the same task usually show good locality (i.e., requesting the data at the locations close to each other on the disk), if there are requests from the same task arriving, the scheduler may choose to schedule these requests, even when there are requests from other tasks arriving earlier. It switches to serve the requests from other tasks when the waiting period expires and there are not requests from the same task. Compared to work-conserving I/O schedulers, non-work-conserving schedulers can improve I/O throughput by exploiting locality. The length of waiting periods is selected to balance improved locality and the utilization of I/O devices. To enforce fairness between I/O tasks, an I/O request scheduler controls the distribution of disk time among the tasks.

2.3 Performance Issues Caused by I/O Inactivity

We use experiments to show that serious performance issues will be caused by I/O inactivity. Specifically, we use SysBench [7] to test I/O throughput in three settings. In the *Bare-metal* setting, we run SysBench on the host. In the *No sharing* setting, we run SysBench in a VM; the VM is the only VM in the host; In the *Vanilla* setting, we consolidate 2 VMs on the same host. In the experiments, each VM has 4 vCPUs, and the host has 4 cores. Thus, in the *No-sharing* setting, there is one vCPU on each core, and in the *Vanilla* setting, each core is time-shared by 2 vCPUs. The VMs are configured to have the same I/O bandwidth quota in KVM [30]. In each VM, the CPU workload in SysBench [7] is run as a compute-bound task, and keep the vCPUs always busy. Note that we select these



(a) I/O bandwidth of individual VM

(b) I/O bandwidth of overall vitualized systems

(c) Fairness of I/O Scheduler

Figure 2: ***Three performance issues caused by I/O inactivity*. "Bare-metal" means physical server; "No sharing" means only one VM running on the host; "Vanilla" means two VMs consolidated and managed by vanilla KVM on one host.**

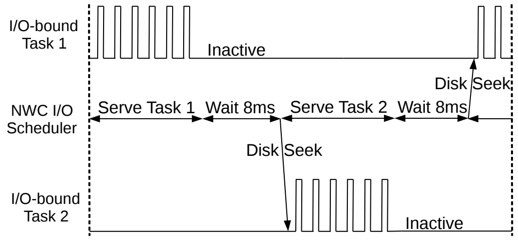


Figure 3: ***I/O inactivity makes Non-Work-Conserving (NWC) I/O scheduler used in VMM ineffective and inefficient because costly disk seeks and waiting time cannot be effectively reduced.***

workloads and settings mainly to ease the demonstration and analysis of the performance issues. Our evaluation with real workloads and normal settings (§5) show that these performance issues can actually be more severe.

Figure 2 (a) and Figure 2 (b) show that I/O inactivity significantly reduces I/O throughput in two different ways. In the experiment shown in Figure 2 (a), we run only one instance of I/O bound task (i.e., I/O workload of SysBench). Among the three settings, *No sharing* has roughly the same I/O throughput as Bare-metal; but the I/O throughput in the *vanilla* setting is about half of those of the other two settings. This is because the VM running the I/O bound task only obtains 50% of CPU time on each core. Thus, the I/O bound task is only active for 50% of the time, as illustrated in Figure 1.

In the experiment shown in Figure 2 (b), we run two instances of I/O bound task, one in each VM. For brevity, we refer to the I/O bound task in the first VM as I/O bound task 1, and refer to the task in the other VM as I/O bound task 2. The bars in the figure show the throughputs of these tasks, as well as the total I/O bandwidth. As shown in the figure, in the *Vanilla* setting, the total throughput drops by 72.1% compared to bare-metal and no sharing, which is more than 50%.

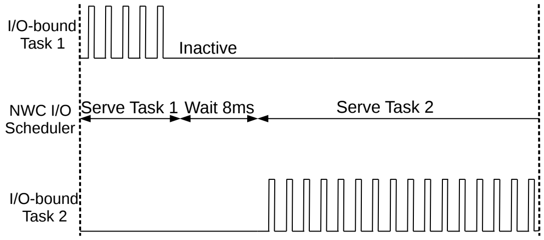


Figure 4: ***I/O inactivity causes unfairness issue.* The I/O task in a VM with more CPU time gets more I/O bandwidth.**

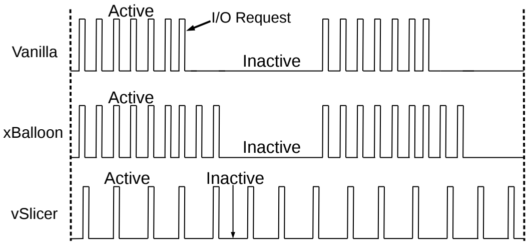


Figure 5: ***The workflow of vanilla, xBalloon and vSlicer.* xBalloon and vSlicer still experience frequent I/O inactivity periods.**

Figure 3 explains the reason. The non-work-conserving I/O scheduler in the VMM serves an I/O bound task in a VM for a short period before the vCPU running the task is descheduled. Then, it waits for 8ms without seeing any requests from I/O bound task 1. Thus, it has to switch and start to serve the I/O-bound task in the other VM (i.e., I/O-bound task 2). The changes between tasks are caused by I/O inactivity. They incur costly disk seeks. The wasted waiting time further reduces I/O throughput.

Figure 2 (c) illustrates the unfairness issue caused by I/O inactivity. It shows that two I/O bound tasks on two VMs with the same I/O priority achieve quite different I/O throughputs because the two VMs are assigned with different CPU time shares. In the experiments, for the *Vanilla* setting, we launch two VMs with the same I/O priority, and run an instance of I/O bound task on each of the VMs. We assign to the VMs with 20% and 80% of CPU time, respectively. For the *Bare-metal* setting and *No sharing* setting, we launch two instances of I/O bound task on the host and the VM, respectively. The two instances of I/O bound task are assigned with the same I/O priority but different CPU time shares (20% and 80%, respectively).

As shown in the figure, the two I/O bound tasks achieve similar I/O throughputs in the *Bare-metal* and *No sharing* settings. However, in the *Vanilla* setting, the I/O bound task in the VM with a larger CPU time share achieves a much higher (5.8x) throughput than that of the I/O bound task in the other VM. Figure 4 explains the cause of this fairness issue. Since VM1 is allocated much less CPU time than VM2, it experiences much longer I/O inactivity periods. As a result, the I/O scheduler serves VM2 for much longer time than VM1.

There are two approaches that may be used to improve I/O throughput. One approach [48, 10, 49] uses smaller time slices (e.g., vSlicer), such that vCPUs are scheduled more frequently, and thus become more responsive to I/O events. As shown in Figure 5 with the curve labeled with vSlicer, this approach reduces the length of each vCPU inactivity period. But I/O inactivity periods become more frequent, and the portion of time in which an I/O task is inactive may not be reduced. Moreover, vSlicer incurs frequent context switches between vCPUs and increases the associated overhead. The other approach [31, 44] lifts the priority of I/O tasks. For example, xBalloon controls how vCPUs consume time slices such that more CPU time can be reserved for the execution of I/O bound tasks on the vCPUs. While this actually lengthens I/O active periods, as shown in Figure 5 with the curve labeled with xBalloon, vCPUs still must be descheduled when they run out of time slices, and I/O inactivity problems are still incurred.

3 vMigrater Design

In this section, we first introduce the design principles and overall architecture of vMigrater. Then, we present the design details of each key component, focusing on how vMigrater monitors the scheduling and descheduling of vCPUs to keep track of their time slices (§3.2), quickly detects I/O-bound tasks (§3.3), and migrates I/O-bound tasks with low overhead (§3.4). Finally, we analyze the performance potential of vMigrater (§3.5).

3.1 Design Principles and Overall Architecture

The design of vMigrater follows three principles:

- Fair: the design of vMigrater must not affect vCPU scheduling (e.g., allocating more CPU time to the vCPUs running I/O tasks) or I/O scheduling in the VMM to avoid any potential unfairness between VMs.
- Non-intrusive: For the wide adoption of vMigrater, the design must be non-intrusive. It should minimize or avoid the modifications to VMM and guest OSs, and should be transparent to applications. Thus, we choose to design vMigrater in the user space of guest OSs. This also helps maintain the original vCPU scheduling and I/O scheduling decisions of the VMM. However, this poses a few challenges, since the migration of I/O bound tasks relies on some key information about vCPU scheduling (e.g., remaining time slice of a vCPU), which is not easy to obtain in the user space.
- Low overhead: vMigrater needs to migrate I/O bound tasks. Frequent migrations may incur high overhead. The design of vMigrater must effectively control the frequency of migrations.

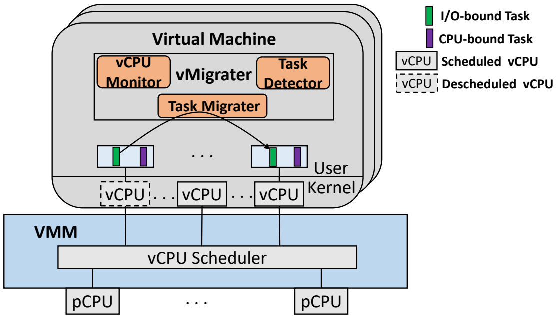


Figure 6: ***Overall Architecture of* vMigrater.**

Figure 6 shows the overall architecture of vMigrater and its position in the software stack. vMigrater resides in each VM, and runs at the user level. Following the above principles, three key components are designed as follows.

vCPU Monitor (§3.2) monitors the scheduling and descheduling of vCPUs. The objective is to measure time slice lengths for each vCPU and use the lengths to predict whether a vCPU is about to be preempted. The prediction is then used to make decisions on when an I/O bound task should be migrated and where it should be migrated.

Task Detector (§3.3) detects I/O activities to quickly determine whether a task is I/O-bound.

Task Migrater (§3.4) makes migration decisions and actually migrates I/O-bound tasks. It makes migration decisions based on the vCPU scheduling information from the Task Migrater and I/O activities of the tasks from the Task Detector. Specifically, it tries to migrate an I/O bound task detected in the Task Detector when the vCPU running the task is about to be descheduled. It migrates the task to another vCPU which may not be descheduled in near future.

3.2 vCPU Monitor Design

The vCPU Monitor uses a heartbeat-like mechanism to detect whether a vCPU is running or has been descheduled, with timer events being heartbeats. The idea is that, when a vCPU is descheduled, it cannot process timer events, and the heartbeat pauses. Specifically, vCPU Monitor runs a sleeping thread, namely vCPU Monitor thread, on each vCPU. The sleeping thread is woken up by a timer periodically. When it is woken up, it checks the current clock time, and compare the time with the time it observes last time. A time difference longer than the period for waking up the thread indicates that the vCPU was descheduled earlier, and has just been rescheduled.

This mechanism is as shown in Figure 7. The vCPU Monitor thread can detect that the vCPU is rescheduled at time t_2 and time t_6. The thread keeps track of the timestamps when the vCPU is rescheduled (e.g., t_2 and t_6) and the timestamps immediately before them (e.g., t_1 and t_5). The time slice lengths can be estimated from these timestamps (e.g., $t_5 - t_2$).

Note that, since a vCPU may be scheduled or descheduled while its vCPU Monitor thread is sleeping, the exact time of the vCPU being rescheduled/descheduled cannot be obtained, and thus accurate time slice lengths cannot be measured with this method. Waking up the vCPU Monitor thread more frequently improves the accuracy of estimation; but it increases the overhead at the same time. Considering that typical time slice lengths are tens of miliseconds, vMIGRATER sets the length of the periods for waking up vCPU Monitor threads to 300 μs to make a trade-off between accuracy and overhead.

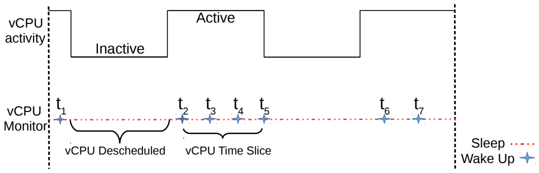


Figure 7: ***vCPU Monitor workflow*.**

3.3 Detecting I/O bound Tasks

Some applications have bursty I/O operations. Thus, vMIGRATER needs to quickly respond to workload changes in each application. vMIGRATER migrates an application when it becomes I/O bound, and stops migration when its I/O phase finishes. However, traditional methods (e.g., Linux top [45] and iotop [26]) for detecting tasks' I/O utilization usually take long time (e.g., seconds). Using these methods may miss the I/O phases in such applications. For instance, the time for an SSD to handle 100MB sequential read is only 100ms. Thus, a much faster method for detecting I/O bound tasks is needed.

vMIGRATER uses an event-driven method to detect I/O-bound tasks quickly. This method monitors the I/O events triggered by I/O requests, and collects the time spent on processing these I/O events. vMIGRATER periodically calculates the fraction of time spent on processing I/O events. (The duration of each period in our design is 5 milliseconds.) It determines that a task becomes I/O bound when the fraction exceeds a threshold.

3.4 Migrating I/O bound Tasks

Task Migrater relies the information from vCPU Monitor and Task Detector to make migration decisions. It first needs to decide which I/O tasks should be migrated. To minimize the overhead, Task Migrater only migrates I/O bound tasks when vCPUs running them are to be descheduled shortly. To find these tasks, Task Migrater estimates the remaining time slice for each vCPU[1]. If the remaining time slice is shorter than the length of two periods for waking up vCPU Monitor threads (i.e., 600 μs)[2], Task Migrater determines that the vCPU is about to be descheduled. Task Migrater then checks the tasks scheduled on the vCPU. If there is an I/O bound task reported by Task Detector, Task Migrater migrates the task.

Second, Task Migrater needs to decide which vCPU the I/O bound tasks should be migrated to. A naïve approach is to migrate I/O tasks to the vCPU with the longest remaining time slice. However, this method has two problems if Task Migrater needs to migrate multiple I/O bound tasks: (1) the I/O bound tasks are migrated to the same vCPU and cannot make progress concurrently; (2) the vCPU might be overloaded by accepting all these tasks, and the performance of its existing tasks is degraded.

Task Migrater migrates I/O tasks to vCPUs in a globally balanced way. Specifically, Task Migrater ranks active vCPUs based on the lengths of their remaining time slices, and ranks the I/O bound tasks to be migrated based on their I/O load levels. It migrates the I/O bound tasks with heavier I/O load levels to the vCPUs with longer remaining time slices. This migration mechanism can prevent the above problems because it distributes I/O bound tasks among active vCPUs. At the same time, it helps maintain high I/O throughput because the

[1] The remaining time slice of a vCPU at a moment (e.g., t_7 in Figure 7) is estimated using the length of time slice assigned to the vCPU before the most recent descheduling of the vCPU (e.g., $t_5 - t_2$) and the CPU time that has already been consumed by the vCPU after the most recent rescheduling of the vCPU (e.g., $t_7 - t_6$).

[2] This is to tolerate the inaccuracy in the estimation of time slices and remaining time slices.

tasks with the most I/O activities are scheduled on the vCPUs that are least likely to be descheduled shortly.

3.5 Performance Analysis

We use Equation (1) to show the performance potential of vMIGRATER. For simplicity, we assume each VM has at least one active vCPU at any given time. Thus, an I/O application can be kept active with vMIGRATER, except when it is being migrated.

$$Speedup_{vMigrater} = \frac{T_{ns} \times N}{T_{ns} + N_{migrate} \times C_{avg}} = \frac{N}{1 + \frac{N_{migrate} \times C_{avg}}{T_{ns}}} \quad (1)$$

Equation (1) calculates the speedup of an I/O application with vMIGRATER relative to its execution without vMIGRATER on a VM. N is the number of vCPUs consolidated on each pCPU (i.e., consolidation rate). T_{ns} is execution time of the I/O application on a VM when its execution is not affected by I/O inactivity problem. This can be achieved by running the application on a vCPU with a dedicated pCPU. It reflects the best performance that an I/O application can achieve on a VM. $N_{migrate}$ is the number of migrations conducted by vMIGRATER. C_{avg} is the average time cost incurred by each migration.

The numerator of equation (1) is the execution time of an I/O application on a VM without vMIGRATER. With N vCPUs consolidated on a pCPU, in each period of N time slices, the I/O application can be active only for a period of one time slice. Thus, its execution time is roughly $N \times T_{ns}$. The denominator is the execution time with vMIGRATER, which is determined by the time spent on application execution and the time spent on migration.

Equation (1) shows that $N_{migrate}$ must be reduced to improve the performance of vMIGRATER. Suppose vMIGRATER migrates the I/O application by a minimum number N_{min} of times in an optimal scenario. Thus, $N_{min} = T_{ns}/T_{ts}$, where T_{ts} is the length of a time slice allocated to a vCPU. In this optimal scenario, the I/O application is moved to a vCPU when the vCPU is just rescheduled; it stays there until the timeslice of the vCPU is used up; it is then moved to another vCPU which is newly rescheduled.

Replacing T_{ns} with $T_{ts} \times N_{min}$ in equation (1), we get:

$$Speedup_{vMigrater} = \frac{N}{1 + \frac{N_{migrate} \times C_{avg}}{N_{min} \times T_{ts}}} \quad (2)$$

Equation 2 shows that the speedup is determined by N and $\frac{N_{migrate} \times C_{avg}}{N_{min} \times T_{ts}}$; N, C_{arg}, and T_{ts} are constants for an application. We denote $\frac{N_{migrate}}{N_{min}}$ as $P_{vMigrater}$, which has a value greater than 1. The speedup is mainly determined by $P_{vMigrater}$. When $P_{vMigrater}$ approaches to 1, the speedup approaches to N. Our experiments show that the speedup with vMIGRATER matches the speedup calculated by Equation 1.

4 Implementation Details

We have implemented vMIGRATER on Linux. The implementation of vCPU Monitor relies on a reliable and accurate clock source to generate timer events. The traditional system time clock cannot satisfy this need when vCPUs time-share a pCPU [44]. Instead, we use the clock source CLOCK_MONOTONIC [18], which is more reliable and can provide more accurate time measurement. The implementation of Task Detector leverages BCC [11, 24] to monitor I/O requests. BCC is a toolkit supported by Linux kernel for creating efficient kernel tracing and manipulation programs.

The implementation of Task Migrater uses two mechanisms, PUSH and PULL, to migrate tasks. A PUSH operation is conducted by the source vCPU of a task to move the task to the destination vCPU, while a PULL operation is initiated by the destination vCPU to move a task to it from the source vCPU. Usually PUSH operations are used. PULL operations are only used when source vCPUs are descheduled and cannot conduct PUSH operations. vMIGRATER's source codes are available on `github.com/hku-systems/vMigrater`.

5 Evaluation

Our evaluation is done on a DELL™ PowerEdge™ R430 server with 64GB of DRAM, one 2.60GHz Intel® Xeon® E5-2690 processor with 12 cores, a 1TB HDD, and a 1TB SSD. All VMs (unless specified) have 12 vCPUs and 4GB memory. The VMM is KVM [30] in Ubuntu 16.04. The guest OS in each VM is also Ubuntu 16.04. The length of a vCPU time slice is 11ms, as recommended by Red Hat [41]. The I/O scheduler in VMM is CFQ with wait time set to 8ms, as recommended by Red Hat [42, 40].

We evaluate vMIGRATER using a collection of micro-benchmarks and 7 widely used applications. Micro- benchmarks include *SysBench [7] sequential read*, *SysBench random read*, and *bursty read implemented by us*. As summarized in Table 1, applications include *HDFS* [5], *LevelDB* [3], *MediaTomb* [9], *HBase* [2], *PostMark* [6], *Nginx* [37], and *MongoDB* [35]. To be close to real-world deployments, PostMark is run with ClamAV (antivirus program) [17] to generate the workload of a complete mail server with antivirus support; LevelDB and MongoDB are deployed as the back-end storage of a Spark [52] system.

Application	Workload
HDFS	Sequential read 16GB with HDFS TestDFSIO [25].
LevelDB	Random scan table with db_bench [4].
MediaT	Concurrent requests on transcoding a 1.1GB video.
HBase	Random read 1GB with HBase PerfEval [25].
PostMark	Concurrent requests on a mail server.
Nginx	Concurrent requests on watermarking images [1].
MongoDB	Sequential scan records with YCSB [8].

Table 1: ***7 applications and workloads.***

Most of the experiments are conducted with the SSD. Only the experiments in §5.4 (fairness of I/O scheduler) use the HDD, because they need a non-work-conserving I/O scheduler (e.g., CFQ) and CFQ is used in Linux to schedule HDD requests.

We compare vMigrater with two related solutions: xBalloon [44] and vSlicer [49]. Because they do not have open-source implementations, we implemented them based on the description in their papers.

Our evaluation aims to answer the following questions:

§5.1: Is vMigrater easy to use?

§5.2: How much performance improvement can be achieved with vMigrater, compared with vanilla KVM and two related solutions? What is the overhead incurred by vMigrater.

§5.3: What is vMigrater's performance when the workload in a VM varies over time?

§5.4: Can vMigrater help the I/O scheduler in the VMM to achieve fairness between VMs?

5.1 Ease of Use

With vMigrater, all 7 real applications we evaluated could run smoothly without any modification. When we evaluate these applications, vMigrater runs in the user-level of the guest OS. There is no need to change any parts of the VM or the VMM.

5.2 Performance Improvements

We first demonstrate that vMigrater can greatly improve the throughput of I/O intensive applications in each VM. For this purpose, we vary the number of VMs hosted on the server from 1 to 8, and run the workload with the micro-benchmarks and the workloads with the real applications summarized in Table 1. In each experiment, we run one instance of the workload in each VM. So the co-located VMs have the same workload. We measure the throughputs of the benchmarks and real applications. When only one VM is hosted on the server, the I/O inactivity problem does not happen; the benchmarks and applications achieve the highest performance. We refer to this setting as **No sharing**, and use the performance under this setting as reference performance. We normalize the performance under other settings (i.e., 2/4/8 VMs consolidated on the server) against the reference performance, and show the normalized performance. Thus, the normalized performance of 1 is the best performance that can be achieved. The closer the normalized performance is, the better the performance is.

Figure 8 shows the normalized throughputs for micro-benchmarks when the number of consolidated VMs is varied from 2 to 8. With vMigrater, the benchmarks consistently achieve better performance than they do on vanilla KVM. At the same time, the performance advantage with vMigrater becomes more prominent when more VMs are consolidated. On average, with vMigrater, the throughputs of these benchmarks are improved by 97%, 225%, and 431% than those on vanilla KVM for the settings with 2 VMs, 4 VMs, and 8VMs, respectively.

Similar performance improvements are also observed with real applications, as shown in Figure 9. On average, with vMigrater, the throughputs of these applications are improved by 72%, 192%, and 342% than those on vanilla KVM for the settings with 2 VMs, 4 VMs, and 8VMs, respectively.

Compared to vSlicer and xBalloon, the applications can also achieve better performance with vMigrater. As shown in Figure 9, On average, with vMigrater, the throughputs of these applications are improved by 88.41%, 74.86%, and 121.22% than those with vSlicer for the settings with 2 VMs, 4 VMs, and 8VMs, respectively; and the throughputs are improved by 3.29%, 83.78%, and 175.37% than those with xBalloon under these three settings.

While vMigrater can significantly improve the throughput of I/O applications when all the consolidated VMs are equipped with vMigrater. Since vMigrater is designed at the user space, it is possible that not all the VMs have vMigrater deployed. We wonder whether vMigrater can still effectively improve I/O throughput in this scenario. To answer this question, we run the workloads with HDFS and LevelDB in one VM and enables vMigrater in this VM; in other colocated VM(s), we run the IS benchmark in NPB benchmark suite [36], and disable vMigrater in the VM(s). Figure 10 shows that the effectiveness of vMigrater is not affected. On average, with

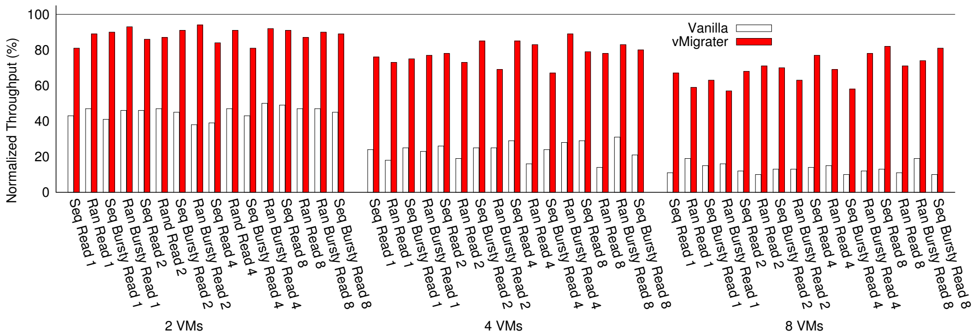


Figure 8: ***Normalized throughputs of micro-benchmarks***

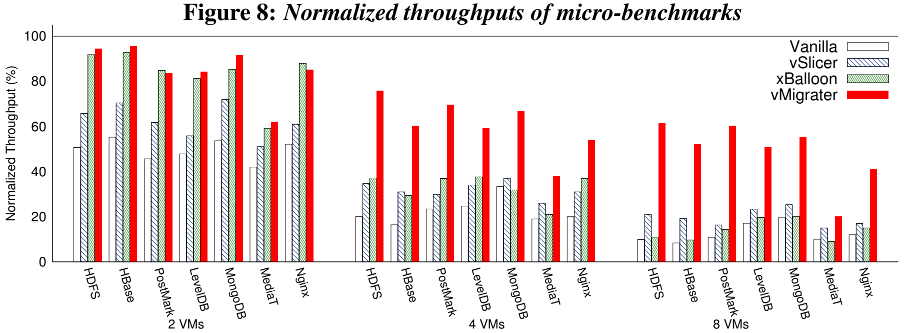


Figure 9: ***Normalized throughput of real applications***

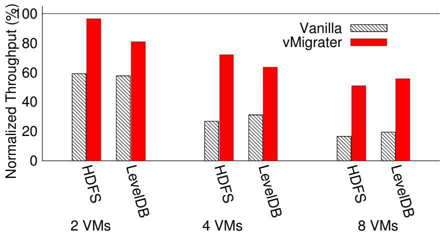


Figure 10: ***Normalized throughput of HDFS and LevelDB when* VMIGRATER *is enabled in one of the consolidated VMs.***

VMIGRATER, the throughputs of these applications are improved by 62.72%, 176.92%, and 218.75% than those on vanilla KVM for the settings with 2 VMs, 4 VMs, and 8VMs, respectively.

To understand how the performance improvements are achieved with VMIGRATER, we profile the executions of the real applications. We collect the number of migrations and the time during which I/O bound tasks "run" on descheduled vCPUs (i.e., I/O inactivity time). We show the data in Table 2 and Table 3.

VMIGRATER greatly improves application performance by first dramatically reducing I/O inactivity time. As shown in Table 2, on average, for the applications, VMIGRATER reduces I/O inactivity time by 860.27%, 657.87%, 562.92%, respectively, relative to vanilla KVM, vSlicer, and xBalloon.

When I/O inactivity time has been dramatically reduced, as we have analyzed in Section 3.5, VMIGRATER maintains high throughputs by minimizing the time spent on migrating tasks, which is determined by the number of migrations and the time to finish each migration. As shown in Table 3, for most applications, the $P_{vMigrater}$ values are very close to 1. This confirms that the migration mechanisms in VMIGRATER are well designed. On one hand, they have effectively migrated I/O bound tasks to keep them active and minimize I/O inactivity. On the other hand, they only migrate the tasks for close-to-minimal times, so as to keep the time spent on migration low. We notice that the $P_{vMigrater}$ value is the highest (1.34) for `MediaTomb` among these applications, and its Speedup is the lowest (1.41). This confirms our performance analysis in Section 3.5.

We also notice that the effectiveness of VMIGRATER slightly reduces when the consolidation rate increases. This is caused by the special design with the vCPU scheduler in KVM (i.e., CFS in Linux), which allocate smaller time slices with higher consolidation rates. This reduces the opportunity to migrate I/O bound tasks. This problem can be mitigated by waking up Task Detector threads more frequently.

Figure 11 shows the response time of the three systems normalized to no sharing. For 7 applications, the response times of VMIGRATER and xBalloon are almost the same. Since each VM has 50% CPU resource, xBalloon has good performance (mentioned above). For `MediaTomb`, all three systems incur high response time because `MediaTomb` combines I/O and compute in one task.

Figure 12 shows three systems' overhead to co-

Application	Vanilla	vSlicer	xBalloon	vMigrater	Ratio
HDFS	121.82s	92.91s	75.27s	6.62s	18.39
LevelDB	129.45s	101.55s	79.84s	17.86s	7.25
HBase	98.13s	69.37s	75.71s	18.93	5.19
MongoDB	39.49s	30.34s	40.57s	3.49s	11.31
PostMark	225.32s	168.01s	113.01s	12.92s	17.44
MediaTomb	108.61s	89.46s	116.96s	34.95s	3.11
Nginx	59.15s	61.72s	42.37s	8.03s	7.37

Table 2: *I/O inactivity time (seconds) of 7 applications.* Four VMs are used. The last column is the ratio between the I/O inactive time with vanilla KVM and that with VMIGRATER.

Application	$N_{migrate}$	N_{min}	$P_{vMigrater}$	Speedup
HDFS	3363	3181	1.05	1.86
LevelDB	2154	2003	1.07	1.75
HBase	3454	3181	1.08	1.76
MongoDB	1545	1363	1.13	1.70
PostMark	5181	4818	1.07	1.82
MediaTomb	2454	1818	1.34	1.41
Nginx	4181	4090	1.02	1.73

Table 3: VMIGRATER *only migrates I/O bound tasks for close-to-minimal times.* Two VMs are used.

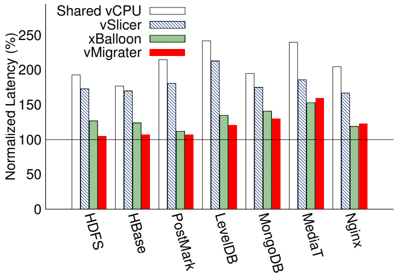


Figure 11: *Response time normalized to "no vCPU sharing".* Two 12-vCPU VMs share 12 pCPUs.

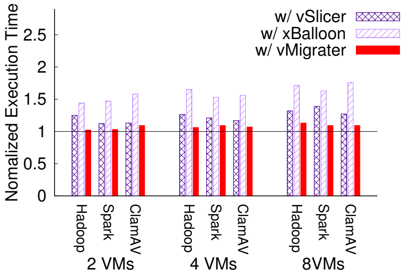
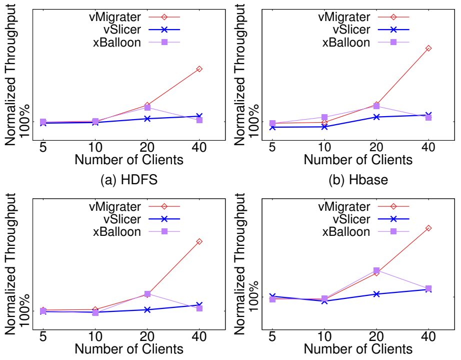


Figure 12: *Execution time normalized to "Vanilla".* "Hadoop" means each VM is running Hadoop standard TeraSort workload; "Spark" means each VM is running standard WordCount workload; "ClamAV" means each VM is scanning virus for the whole OS. Each VM has 12 vCPUs.

running compute-bound applications in the same VM. xBalloon's overhead is much higher than VMIGRATER and vSlicer because xBalloon prioritizes I/O-bound tasks and delays compute-bound tasks. vSlicer's overhead is higher than VMIGRATER because it incurs much more context switching overhead for compute-bound tasks. Unlike xBalloon and vSlicer, VMIGRATER almost would not delay co-running applications (§3).

5.3 Robustness to Varing Workloads



(a) HDFS

(b) Hbase

(c) PostMark

(d) MongoDB

Figure 13: *Throughput scalability on the loads of VMs, normalized to vanilla.* Each client consumes around 20% CPU resources; two 2-vCPU VMs share two pCPUs; the more concurrent clients, the more faster VMIGRATER than vSlicer and xBalloon.

Figure 13 shows the three systems' throughput under varing workloads. When the number of clients is lower than 10, the throughput of VMIGRATER is almost the same as vSlicer and xBalloon because VMs are not shared. VMIGRATER is not started when there is no sharing. As the number of clients increased to 40, VMIGRATER outperforms the other two systems significantly because VMIGRATER can efficiently avoid I/O inactivity periods by migrating I/O tasks to scheduled vCPUs. vSlicer and xBalloon do not work when vCPU is inactive. xBalloon has almost the same performance as VMIGRATER for around 20 clients (each VM has 50% CPU resource). However, VMIGRATER is much more scalable than vSlicer and xBalloon when workloads increase.

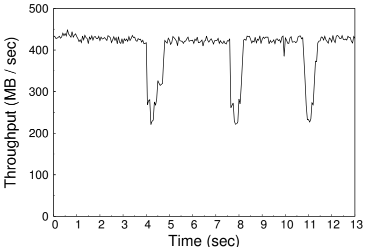
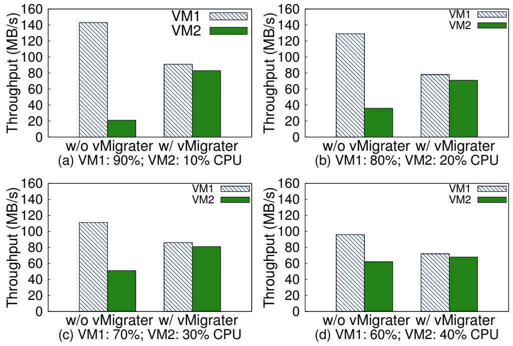


Figure 14: VMIGRATER*'s performance on handling the load change of vCPUs by adding clients dynamically.* 8 clients at the time 0; each client exhausts 20% CPU resource; two 2-vCPU VMs share two pCPUs.

Figure 14 shows the robustness of VMIGRATER in the face of suddenly changing workloads. There are 8 clients at 0s, and the VMs are not overloaded. At around 4s, 8s and 11s , 4, 8 and 16 more clients are added, VMIGRATER's throughput decreases to around 240MB/s, but it becomes stable (peak, around 430MB/s) again after a short period because VMIGRATER needs some time to precisely re-estimate the time slices of vCPUs and then migrate the I/O bound tasks (§7).

5.4 Fairness for I/O Scheduler



(a) VM1: 90%; VM2: 10% CPU

(b) VM1: 80%; VM2: 20% CPU

(c) VM1: 70%; VM2: 30% CPU

(d) VM1: 60%; VM2: 40% CPU

Figure 15: vMigrater *improves the fairness of I/O Scheduler.* Two 12-vCPU VMs share 12 pCPUs; each VM is allocated different CPU resources but the same I/O bandwidth.

Figure 15 shows the fairness of the VMM I/O scheduler among VMs. In Figure 15 (a), (b), (c) and (d), VM1's CPU resource decreases from 90% to 60%, and VM2's CPU resource increases from 10% to 40%. Each VM runs `TestDFSIO` (I/O-bound task) and `TeraSort` (compute-bound task) concurrently, and each VM is allocated with the same I/O bandwidth. Without vMigrater, `TestDFSIO` throughput is related to the CPU resource allocated to the VM, which shows that vanilla hurts the fairness of the I/O scheduler in the host OS. With vMigrater, two VMs in each figure achieve roughly the same `TestDFSIO` throughput, which implies vMigrater maintains fairness (roughly the same I/O bandwidth) for the two VMs.

6 Related Work

Shortening time slices. Many efforts have focused on shortening the time slices of vCPUs [10, 49, 48] for vCPUs to process I/O requests more frequently. This solution has two drawbacks: (1) the I/O inactivity period still exits and could degrade I/O performance; (2) it suffers from performance degradation because of frequent context switches [21, 46, 32]. [48] uses the same idea to reduce the delay of IRQ processing. These solutions require intensive modifications to both the VMM and guest OS kernel.

Dedicating CPUs. Dedicating CPUs [43, 14] aims to solve the resource contention problem. This solution makes fewer vCPUs share one pCPU in order to reduce contention. These systems are complementary to vMigrater because they focus on reducing the vCPU sharing, while vMigrater focuses on improving performance in the shared setting.

Task-aware Priority Boosting. Existing systems [21, 31, 15, 44, 39, 20, 29, 50, 23, 34, 22, 33, 16, 28] focus on prioritizing latency-sensitive tasks to improve overall performance. Task aware VM scheduling [31] improves the performance of workloads by prioritizing I/O bound VMs. [31] works in the VMM layer and may require changing the source codes of the host OS. xBalloon preserves the priority of I/O tasks by preserving CPU resource for I/O tasks. However, the vCPUs are still descheduled so the I/O inactivity periods still exist. xBalloon works best for VMs with single vCPUs, while vMigrater is designed for multi-vCPU VMs.

7 Conclusion and Future Work

This paper identifies I/O inactivity problem in VMs which has not been adequately studied before. It presents vMigrater, a simple, fast and transparent system that can greatly mitigate I/O inactivity.

vMigrater has two limitations, and we leave them as future work. First, when vMigrater runs in a VM, the performance of an application in the VM may drop temporarily when the application's workload changes suddenly. Our evaluation (see §5.3) shows that, when the number of clients for HDFS increased from 16 to 32, HDFS's throughput dropped by 45.6% for 1.3 seconds and then went back to the peak throughput immediately. The reason is that the sudden changing workload makes time slices of some vCPUs not stable, and vMigrater needs some time to precisely re-estimate the time slices of vCPUs and then migrate the I/O bound tasks. Second, vMigrater mainly aims to mitigate the performance degradation caused by disk (HDD or SSD) I/O inactivity periods in VMs, and it is not designed to handle network I/O. Comparing to disk I/O, network I/O is much more sparse, and we have not come across any situation where vMigrater affects the performance of network I/O in our evaluation.

Acknowledgments

We thank anonymous reviewers for their helpful comments. This work is funded in part by the US National Science Foundation under grants CCF 1617749 and CNS 1409523, research grants from Huawei Innovation Research Program 2017, HK RGC ECS (27200916), HK RGC GRF (17207117), HK CRF grant (C7036-15G), and a Croucher innovation award.

References

[1] Adding watermarks to images using alpha channels. http://php.net/manual/en/image.examples-watermark.php.

[2] HBase. https://hbase.apache.org/.

[3] LevelDB. https://github.com/google/leveldb.

[4] LevelDB Benchmarks. http://www.lmdb.tech/bench/microbench/benchmark.html.

[5] The Hadoop Distributed File System. http://hadoop.apache.org/hdfs/.

[6] The PostMark Benchmark. http://www.filesystems.org/docs/auto-pilot/Postmark.html.

[7] SysBench: a system performance benchmark. http://sysbench.sourceforge.net, 2004.

[8] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB, 2004.

[9] MediaTomb - Free UPnP MediaServer. http://mediatomb.cc/, 2014.

[10] J. Ahn, C. H. Park, and J. Huh. Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 394–405. IEEE Computer Society, 2014.

[11] https://iovisor.github.io/bcc/.

[12] https://en.wikipedia.org/wiki/CFQ.

[13] https://en.wikipedia.org/wiki/Completely_Fair_Scheduler.

[14] L. Cheng, J. Rao, and F. Lau. vscale: automatic and efficient processor scaling for smp virtual machines. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 2. ACM, 2016.

[15] L. Cheng and C.-L. Wang. vbalance: using interrupt load balance to improve i/o performance for smp virtual machines. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 2. ACM, 2012.

[16] R. C. Chiang and H. H. Huang. Tracon: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 47. ACM, 2011.

[17] http://www.clamav.net/.

[18] http://man7.org/linux/man-pages/man2/timer_create.2.html.

[19] https://en.wikipedia.org/wiki/Deadline_scheduler.

[20] X. Ding, P. B. Gibbons, and M. A. Kozuch. A hidden cost of virtualization when scaling multicore applications. In *HotCloud*, 2013.

[21] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, pages 73–84, 2014.

[22] S. Gamage, A. Kangarlou, R. R. Kompella, and D. Xu. Opportunistic flooding to improve tcp transmit performance in virtualized clouds. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 24. ACM, 2011.

[23] S. Gamage, C. Xu, R. R. Kompella, and D. Xu. vpipe: Piped i/o offloading for efficient data movement in virtualized clouds. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[24] B. Gregg. Performance superpowers with enhanced BPF. Santa Clara, CA, 2017. USENIX Association.

[25] Hadoop. http://hadoop.apache.org/core/.

[26] https://linux.die.net/man/1/iotop.

[27] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 117–130. ACM, 2001.

[28] W. J. Jianchen Shan and X. Ding. Rethinking the scalability of multicore applications on big virtual machines. In *IEEE International Conference on Parallel and Distributed Systems*. IEEE, 2017.

[29] A. Kangarlou, S. Gamage, R. R. Kompella, and D. Xu. vsnoop: Improving tcp throughput in virtualized environments via acknowledgement offload. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[30] http://www.linux-kvm.org/.

[31] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for i/o performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 101–110. ACM, 2009.

[32] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, page 2. ACM, 2007.

[33] H. Lu, B. Saltaformaggio, R. Kompella, and D. Xu. vfair: Latency-aware fair storage scheduling via per-io cost-based differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 125–138. ACM, 2015.

[34] H. Lu, C. Xu, C. Cheng, R. Kompella, and D. Xu. vhaul: Towards optimal scheduling of live multi-vm migration for multi-tier applications. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 453–460. IEEE, 2015.

[35] Mongodb. http://www.mongodb.org, 2012.

[36] http://www.nas.nasa.gov/publications/npb.html.

[37] Nginx web server. https://nginx.org/, 2012.

[38] https://en.wikipedia.org/wiki/Noop_scheduler.

[39] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10. ACM, 2008.

[40] Red Hat. What is the suggested I/O scheduler to improve disk performance (2017). https://access.redhat.com/solutions/5427.

[41] RedHat. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/6.0_technical_notes/deployment.

[42] RedHat. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/index.

[43] X. Song, J. Shi, H. Chen, and B. Zang. Schedule processes, not vcpus. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, page 1. ACM, 2013.

[44] K. Suo, Y. Zhao, J. Rao, L. Cheng, X. Zhou, and F. Lau. Preserving i/o prioritization in virtualized oses. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 269–281. ACM, 2017.

[45] http://man7.org/linux/man-pages/man1/top.1.html.

[46] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Proceedings of the 2007 workshop on Experimental computer science*, page 4. ACM, 2007.

[47] VMware. Vmware horizon view architecture planning 6.0. In VMware Technical White Paper (2014).

[48] C. Xu, S. Gamage, H. Lu, R. R. Kompella, and D. Xu. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. In *USENIX Annual Technical Conference*, pages 243–254, 2013.

[49] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu. vslicer: latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 3–14. ACM, 2012.

[50] C. Xu, B. Saltaformaggio, S. Gamage, R. R. Kompella, and D. Xu. vread: Efficient data access for hadoop in virtualized clouds. In *Proceedings of the 16th Annual Middleware Conference*, pages 125–136. ACM, 2015.

[51] Y. Xu and S. Jiang. A scheduling framework that makes any disk schedulers non-work-conserving solely based on request characteristics. In *FAST*, pages 119–132, 2011.

[52] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

Placement of Virtual Containers on NUMA systems: A Practical and Comprehensive Model

Justin Funston[*], Maxime Lorrillere[*], Alexandra Fedorova[*], Baptiste Lepers[†], David Vengerov[‡], Jean-Pierre Lozi[‡], Vivien Quéma[§]

[*]*University of British Columbia,* [†]*EPFL,* [‡]*Oracle Labs,* [§]*IMAG*

Abstract

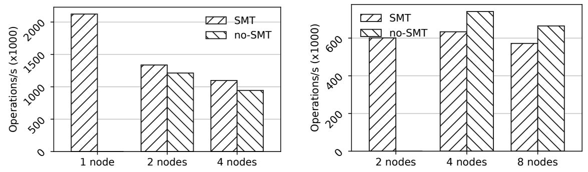
Our work addresses the problem of placement of threads, or virtual cores, onto physical cores in a multicore NUMA system. Different placements result in varying degrees of contention for shared resources, so choosing the right placement can have a large effect on performance. Prior work has studied this problem, but either addressed hardware with specific properties, leaving us unable to generalize the models to other systems, or modeled much simpler effects than the actual performance in different placements.

Our contribution is a general framework for reasoning about workload placement on machines with shared resources. It enables us to build an accurate performance model for any machine with a hierarchy of known shared resources *automatically*, with only minimal input from the user. Using our methodology, data center operators can minimize the number of NUMA (CPU+memory) nodes allocated for an application or a service, while ensuring that it meets performance objectives.

1 Introduction

We address the problem of placing a virtual container on a multicore NUMA system. Hardware resources allocated to a container determine how well it performs and how much energy it consumes. Roughly speaking, there are two decisions affecting which hardware resources are allocated to a container. First, the user decides how many cores, memory and perhaps other resources the container requires. Second, the system software decides, when launching the container, how to map the container's virtual cores onto physical cores. Our work proposes a solution for automatically making this second decision.

The placement of virtual cores onto physical cores can have a large and unpredictable effect on performance. Consider the following experiment, where we use MongoDB's WiredTiger key-value store [3] running a B-tree search using 16 threads. We run this application in a `lxc` container on an Intel NUMA system and on an AMD NUMA system (Fig. 2 provides their overview). Suppose our goal is to maximize the throughput. How should we place this container? Assuming that each virtual core gets its own physical core, how do we place virtual cores onto NUMA nodes? Do we squeeze them onto as few nodes as possible?, do we spread them evenly across all nodes?, or do we use a middle ground?



(a) WiredTiger, Intel

(b) WiredTiger, AMD

Figure 1: Throughput of the WiredTiger key-value store on two NUMA systems.

As Fig. 1 shows the right answer can vary greatly from one system to the next. On the Intel system, the application performs significantly better when all of its threads run on a single node. On the AMD system, four nodes are better than two, only if we do not use SMT, but using eight nodes does not buy you better performance[1].

There is a number of factors responsible for this dissimilar behaviour. When all threads are squeezed into a single NUMA node, they experience more resource sharing: on the Intel system they have no choice but to share the SMT pipeline and the L3 cache. Resource sharing can be contentious (where threads compete for cache space and hardware queues [34]) or cooperative (where threads pre-fetch data for each other [27]). Furthermore, with all the threads running on a single NUMA node,

[1] A single-node configuration for AMD is not shown: we used 16 virtual cores, so we could not fit them onto a single node (with 8 cores) while ensuring that each virtual core gets its own physical core.

cross-thread communication has a lower latency, because it occurs via the L3 cache, as opposed to a slower cross-chip interconnect. Apparently, the benefits of faster communication and cooperative resource sharing outweigh the cost of resource contention on the Intel system, but not on the AMD system.

The best-performing placement for a container, therefore, is difficult to predict. The problem becomes even harder if the goal is to not only pick the best-performing placement, but to achieve a trade-off between the number of used nodes and performance. Our work proposes a solution that works as follows:

Step 1: The user provides a simple abstract specification of the shared resources present on the target hardware. Identifying the right level of abstraction was key to being able to automatically construct models for different target systems. The abstraction we propose in this work is called *scheduling concerns* (§4).

Step 2: Using shared resource specification, our algorithm generates, for a given container size, a list of *important placements* – placements that will likely yield different performance on the target hardware[2]. This step is crucial for automatically training a model: while the total number of potential placements is measured in billions (making training infeasible), the number of important placements is only a couple dozen. We introduce the concept of important placements and propose the algorithms for automatically generating them (§4).

Step 3: Using our script, the user trains a machine learning model for the target hardware and the target number of vCPUs (§5). Unlike prior work, we do not rely on hardware performance events (HPE) as model features. Manual selection of the right HPEs puts too much burden on the user. Automatic selection turned out to be impractical on modern machines with 1000s of HPEs. Instead, our model uses as inputs actual performance measurements obtained in two different placements. This approach makes our model-building methodology easier to port across hardware, reduces the training time and achieves higher accuracy, compared to using HPEs.

Step 4: The scheduler runs the virtual container in two different placements, for a couple of seconds in each, feeds their performance into the model and obtains a vector of predicted performance values in each important placement. Using this vector, the system decides what placement to use and remaps the virtual cores. Since the container needs to run in two placements, its memory may need to be migrated if these placements do not use the same NUMA nodes. We improve on memory migration in Linux and evaluate its overhead in §7.

Our solution enables determining the best placement for a *specific virtual container*, but not how to interleave different containers on the same NUMA node. Some data center operators we spoke to do not interleave containers, others do, so we leave that decision up to the operator. Going back to the scenario in Fig.1, using our tools the scheduling system can quickly decide that on the Intel host it is sufficient to provide a single NUMA node for the key-value store in order to maximize its throughput. Then the remaining nodes can be used to host other containers. We believe that our techniques can be used to build scheduling systems that pack virtual containers onto physical hardware more efficiently.

Our main contribution is ***abstractions and methodology for constructing accurate and portable models for predicting performance of a container in various placements on NUMA systems, regardless of what shared resources are present***. We evaluate it by automatically generating performance models for two different hardware systems and measuring their accuracy using cross-validation (§6). We also present a use case demonstrating how the model could be used in practice (§7).

2 Background and Related Work

Workload placement on multicore systems has been explored for over a decade. Early studies examined contention between single-threaded applications for a specific resource, such as the SMT instruction pipeline [25, 14], or shared caches and memory controllers [34, 12, 21, 31]. Later work extended the techniques to multithreaded workloads and to additional resource combinations, such as SMT and shared caches [33], memory controllers and the shared interconnect [9, 18]. While laying a crucial foundation for our work, these prior techniques did not provide a general solution for reasoning about such systems. For instance, while the work of Zhuravlev et al. [34] showed us how to avoid interference for shared memory controllers and the work of Lepers et al. [18] showed us how to place applications on machines with asymmetric interconnects, we still do not know how to build a model that combines both concerns.

Techniques used in prior work did not allow for automatic combination of several models. Every model required manual design: careful selection of hardware performance events [34, 17, 18, 9, 12] or even manual crafting of artificial "probe" workloads or "Rulers" [31, 33] that must be run side-by-side with the target workloads to determine their sensitivity to contention.

Dwyer used an automated model-building methodology, where automatically selected features (from all HPEs available on the machine) were fed into a variety of machine-learning models [11]. However, the model predicted a rather simple outcome: a performance degradation when a target workload was co-scheduled with an interfering one, and not the performance in different place-

[2] §4 defines the *important placement*.

ments. Consistent with our finding that HPEs observed in a single placement are poor model features, Dwyer's study reported rather poor prediction accuracy.

A recent system, Pandia [15] not only accurately predicts performance of different workload placements on a multicore NUMA machine, but also predicts how an application would perform with different numbers of threads. Unforunately, to make predictions, Pandia requires performance observations of six runs with different thread counts, which is difficult to do online because most real applications cannot easily reconfigure their thread count on demand. Despite addressing many limitations of previous work, fundamentally Pandia still relies on the machine-specific modelling methodology that prevents easily transferring results to other systems. Pandia's authors capture factors that contribute to performance, such as cache contention, latency of communication, and load balancing, in a set of machine-specific equations. If the model had to be adapted to another machine, the equations would have to be manually reformulated.

We believe that investing that much effort into designing new models for every new type of hardware puts an unreasonable burden on system engineers. Instead, we sought a future-proof methodology that uses easily available information about a machine's shared resources and automatically builds an accurate performance model.

There are two recent studies that address a different problem, but use techniques that could be adopted in our work. CherryPick [4] handles cloud configuration options, like CPU count, amount of RAM, disk speed, and network speed, but not multicore resources. CherryPick uses Bayesian optimization to minimize the number of search configurations needed and achieves high accuracy despite the non-linearity of performance. The Bayesian optimization approach could potentially work well with our goal and resources, and is a possible avenue of future work. PARIS [30] is similar to CherryPick in that it handles the same type of resources and its goal is to help cloud customers choose the correct cloud configuration. It does not abstract or handle multi-core resources.

3 Assumptions and Limitations

Identically scored placements yield identical performance. As we explain in §4, a placement is identified by the degree of sharing for each hardware resource, to which we refer as the *score*. A vector of scores identifies a placement. Placements with identical score vectors are deemed to yield identical performance for a given workload. This assumes that our machine model must be informed about all shared resources that might affect performance. Most solutions in this space also assume awareness of all shared resources. A radically different approach would be a statistical technique that searches for an optimally performing placement by trying a sufficient number of random placements [23]. Unfortunately, the best known techniques require trying *thousands* of placements and assume that performance in all placements fits a Generalized Pareto distribution — an assumption that does not hold in our case.

A workload is encapsulated in a virtual container. Data centers use virtualization for a variety of reasons, so this assumption dovetails with our target environment. Managed cloud environments present their offerings as a menu of virtual instances with a fixed number of vCPUs per instance (see [1], for example). As a result, we can feasibly train a separate model for each type of hardware and each vCPU count; we do not have to incorporate the effects of varying number of threads into the model, which would make it more complicated. We are not addressing the problem of finding the optimal number of threads or vCPUs for the workload; for that, users can resort to other tools [15, 26].

A NUMA node is a unit of resource allocation. Our solution predicts the performance of a container in all important placements, provided that the target container does not share NUMA nodes with other containers. Unused NUMA nodes can be safely used to run other containers without interference as long as those nodes do not share the interconnect – a condition that can be automatically checked using the machine specification[3]. Suppose that the "best" number of NUMA nodes chosen for a container gives us more physical cores than the container needs. Then the remaining cores would be left idle if no other containers used them. Some data center operators find this acceptable, reasoning that the cost of leaving cores idle is negligible relative to missing performance targets; others contend that maximizing the utilization of physical cores is very important. Our solution does not dictate the decision. If the operator chose to interleave containers on NUMA nodes, our modeling techniques would need to be extended to provide performance predictions under interleaving. Another alternative would be to only interleave with "safe" containers, e.g., those with low CPU utilization or otherwise known to cause negligible interference. We leave the exploration of these scenarios to future work.

We consider only balanced placements. A balanced placement is one where the number of vCPUs is evenly divisible by any number of shared resource units considered for placement. For instance, if we have shared L3 caches on the system, we will only consider placements where the number of vCPUs sharing each L3 cache is equal. Uneven sharing can cause unpredictable performance effects on the workload, for example by creating

[3] Experimental results confirming this statement are available [13].

stragglers, so we choose to not model these effects.

4 Abstract machine model

A major obstacle to a solution to the placement problem is the sheer number of possible placements. For 16 virtual cores on a 64 core system, the number of possible placements is the combinations of 16 objects chosen from a set of 64, which is on the order of 10^{14}. It is essential to exploit the symmetry in the system to reduce the number of placements to a manageable number. By this we mean that for most types of shared resources it does not matter *which* shared resources are being used but *how much* of the shared resources is available to the workload. For instance, for the workload in Fig. 1 on the AMD system, it does not matter which L3 cache it uses, all that matters is whether it has two, four or eight L3 caches at its disposal.[4]

We tackle this with the concept of ***scheduling concerns***. A single scheduling concern is responsible for a single hardware resource, or an inseparable set of hardware resources that affect the performance of vCPU placements. The primary purpose of a scheduling concern is to provide a numerical ***score*** when given a vCPU placement. The score represents the *static* utilization of the particular resource, meaning that it only depends on the vCPU placement, not the dynamic behavior of a workload. A simple example is an "L2 cache" resource. If in a given placement all the virtual cores share a single L2 cache, the score for the L2 cache scheduling concern will be equal to one. If in another placement the cores are spread over two L2 caches, the score will be equal to two, and so on. So, two placements might use completely different NUMA nodes and physical cores, but if they use the same number of L2 caches then they will both have the same L2 cache score. From the vantage point of the L2 cache, these placements will be *identical* in terms of performance. For non-symmetrical resources, such as the cross-chip interconnect on some systems, instead of counting how many links are used by a placement in order to obtain the score, we would add up the total available bandwidth of all links used by a placement. A vector of numeric scores for all scheduling concerns uniquely identifies each placement that is distinct with respect to sharing of resources. Placements with identical vectors are deemed identical with respect to resource sharing, so we can discard the duplicates when training our model. ***By considering only the placements with distinct score vectors, we substantially reduce the space of relevant placements and make the problem tractable.***

There are two additional pieces of information a scheduling concern needs in order to identify the important placements. The first is whether the concern's score is proportional to the user's cost, which is the case for resources like NUMA nodes because fewer nodes (lower score) means more containers can be packed onto a system. If a lower score for a resource only meant worse performance, we could simply discard placements with a lower score for that resource (all other scores being equal) from our list of important placements. But since we want users to be able to make cost-performance trade-offs, placements with lower scores but potentially lower cost could still be relevant. The second piece of information needed by a scheduling concern is whether the resource encompassed by a concern can ever have an inverse relationship with performance. For some resources, like the L2 cache, a higher score is usually better, but for some workloads such as those showing cooperative cache sharing, a smaller score (using fewer L2 caches) may actually improve performance. For other resources, like the shared interconnect described below, a lower score will never improve performance and would not result in a lower cost for the user, so we can safely ignore placements with lower scores when all else is equal.

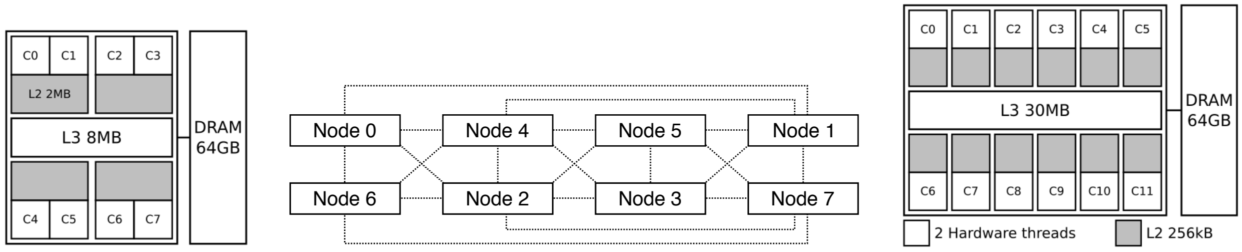
In practice, a single scheduling concern may cover multiple shared resources because some resources are inseparable with respect to thread placement. Threads sharing a physical core via SMT typically share a cache, the instruction front-end, and functional units. In cases like this, a single scheduling concern is still sufficient.

Our AMD system (Fig. 2) has multiple NUMA nodes, an asymmetric interconnect, and a form of SMT. For this system we developed the scheduling concerns shown in Table 1. For the L2/SMT and L3 concerns, the score for a particular placement can be calculated directly from information provided by the operating system. The OS also provides information on the interconnect topology, but it is simpler and more accurate to measure the aggregate bandwidth with a benchmark (e.g. *stream* [20]) for each possible combination of nodes.

For example, for a 16-vCPU container in an eight-node placement without SMT the score vector for the AMD system is [16, 8, 35000], because this placement uses 16 L2 caches (16 hardware threads, one per cache), eight L3 caches (8 nodes) and has an IC bandwidth of 35GB/s. For the same placement, but with SMT, the score vector would be [8, 8, 35000], because on each node two hardware threads would be collocated on the same L2 cache, so we would use half the L2 caches than in the case without SMT.

Each concern is relatively easy to implement, and can be developed independently. Since it does not require a performance expert, we envision the specification of concerns being provided as part of system BIOS. ***Over-***

[4] The exception is asymmetric resources, for instance if one NUMA node is positioned closer to the system NIC than others; our model allows caputring this asymmetry.



(a) AMD Opteron 6272 node

(b) AMD interconnect

(c) Intel Xeon E7-4830 v3 node

Figure 2: The two systems used in our study. The first is a quad AMD Opteron 6272. It has eight NUMA nodes (schematically shown in Figure 2a) connected with an asymmetric interconnect (Figure 2b) and a total of 64 cores. Pairs of cores share the instruction front-end, L2 cache, and floating point units. The second system is a quad Intel Xeon E7-4830 v3 with four NUMA nodes (Figure 2c) and 96 hardware threads (12 physical cores per node with SMT). The interconnect (not shown) is symmetric.

Concern	Score	Resources	Cost?	Inverse Perf Possible?
L2/SMT	Number of L2 caches in use	L2 cache, instruction fetch and decode, and floating point units	Y	Y
L3	Number of L3 caches in use	L3 cache, memory controller, and bandwidth to DRAM	Y	Y
Interconnect	Aggregate bandwidth between nodes in use	Interconnect bandwidth	N	N

Table 1: Scheduling concerns used on our AMD test system (shown in Figure 2).

all, we found scheduling concerns to be a powerful abstraction that enables encoding shared resources on a variety of hardware and makes the model easy to port to new hardware.

Next, from the concerns and hardware topology we need to derive the *important placements*. An important placement must have a score that ensures it satisfies three properties: **(1)** conform to our balanced assumption, **(2)** be feasible: i.e., not assign more than one vCPU to a single hardware thread, and **(3)** not be superseded by a strictly better placement.

Given a score s and the number of vCPUs v, the balance property is encoded as $v \bmod s = 0$, and the feasibility property is encoded as $v/s \leq Capacity$, where capacity is the number of hardware threads available in a single instance of the resource if applicable: e.g. there are eight hardware threads per L3 cache on our AMD test system. We also define the *Count* of a concern as the total number of that resource on the system, so our AMD test system has an *L2Count* of 32 for example. The first step in generating important placements is generating the possible scores that satisfy the balance and feasibility requirements individually. This is done for each scheduling concern that can affect cost or have an inverse relationship with performance. For our AMD test system this step is shown in Algorithm 1.

Now that we have all balanced and feasible placements, and before filtering the duplicates, we need to enumerate all possible placements whose performance the scheduler might want to predict if more than one container were running on the system. For example, suppose that after placing one container onto two nodes on the system, the scheduler might want to place other containers on the remaining nodes, so it should be able to predict the performance on any combination of those nodes. Therefore, we must keep track of possible placements on those remaining nodes in order to properly train the model. The packings are generated with a recursive method shown in Algorithm 2. On our AMD system, we use the L3 scores because the L3 scheduling concern corresponds to NUMA nodes, and NUMA nodes are our unit of resource allocation (see §3).

Next, as shown in Algorithm 3, packings that are duplicates and packings that are not Pareto-efficient with respect to the interconnect score are filtered out (since the interconnect concern does not affect cost and cannot have an inverse relationship with performance). Because the L2 and L3 scores can affect cost or have an inverse relationship with performance, placements are not filtered based on them.

As an example of a Pareto-efficient packing, on our AMD system we need to keep the 4-node placement that

Algorithm 1 Generating possible L2 and L3 scores

```
L3Scores = List()
for $i \leftarrow 1, L3Count$ do
    if $v/i \leq L3Capacity \wedge v$ mod $i = 0$ then
        L3Scores.append(i)
    end if
end for
L2Scores = List()
for $i \leftarrow 1, L2Count$ do
    if $v/i \leq L2Capacity \wedge v$ mod $i = 0$ then
        L2Scores.append(i)
    end if
end for
return L3Scores, L2Scores
```

Algorithm 2 Generating packings of placements

```
Packings = List()
procedure GENPACK(L3Scores, NodesLeft, Current)
    for all $L3S$ in L3Scores do
        if $L3S >$ len(NodesLeft) then
            continue
        end if
        for all $n$ in Combinations(NodesLeft, L3S) do
            Remaining = NodesLeft - $n$
            NewPacking = Current.append($n$)
            if len(Remaining) $> 0$ then
                GenPack(L3Scores, Remaining,
                            NewPacking)
            else
                Packings.append(NewPacking)
            end if
        end for
    end for
end procedure
return Packings
```

uses nodes $\{2,3,4,5\}$ because it is the 4-node placement with the highest interconnect score. Therefore the placement using nodes $\{0,1,6,7\}$ is also an important placement and will be kept because it is the placement that can be packed with the best 4-node placement. Continuing, suppose that we consider a 4-node placement that uses nodes $\{0,1,4,5\}$. If we were to use this placement at runtime, the remaining set of four nodes, potentially used for another workload, is $\{2,3,6,7\}$. Both of these placements have poor interconnect scores, in part because there is a two-hop distance between nodes $\{0,5\}$ and nodes $\{3,6\}$. Instead, we can pack the machine with a better combination of 4-node placements: $\{0,2,4,6\}$ and $\{1,3,5,7\}$. Using this observation, the vectors for placements $\{0,2,4,6\}$ and $\{1,3,5,7\}$ will be kept over the worse pair of 4-node placements.

Algorithm 3 Generating important placements

```
Nodes = range(0, L3Count)
Packings = GenPack(L3Scores, Nodes, List())
Remove duplicates from Packings
for all ($a$,$b$) in Permutations(Packings, 2) do
    if L3 Scores in $a \neq$ L3 Scores in $b$ then
        continue
    end if
    $aIC$ = Sorted interconnect scores of $a$ placements
    $bIC$ = Sorted interconnect scores of $b$ placements
    ToRemove = True
    for $i$ in range(0, len($aIC$)) do
        if $aIC[i] > bIC[i]$ then
            ToRemove = False
        end if
    end for
    if ToRemove then
        Remove $a$ from Packings
    end if
end for
ImportantPlacements = List()
for all Placements $p$ in Packings do
    $n \leftarrow L2Count/L3Count$
    $L3S$ = L3 Score of $p$
    for all L2S in L2Scores do
        if $n \cdot L3S \geq L2S$ then
            ImportantPlacements.append($p$)
        end if
    end for
end for
return ImportantPlacements
```

After this process is complete, we are left with the important placements. For our AMD system we have 13 of them: two 8-node placements (one sharing L2 caches and one not), three 2-node placements (with the best and second-best interconnect score, and one placement used to pack when specific 4-node placements are used), and eight 4-node placements (half sharing L2 caches, half not, and various interconnect scores relevant for packing). Our Intel test system (Fig. 2), on the other hand, only uses an L2/SMT concern and an L3 concern. With 24 virtual cores per container, it has seven important placements which are all of the placements that satisfy the balance and feasibility constraints: a one node placement sharing L2 caches, two 2-node placements, two 3-node placements, and two 4-node placements.

5 Performance Predictions

Automatic model-building techniques learn how to map a set of features describing data to a predicted outcome.

The outcome we would like to model is a vector of performance values in all important placements, relative to a baseline placement. For example, if there are three important placements, and the performance in the second and third is 20% and 30% better than that in the first baseline placement, the performance vector will be: $[1.0, 0.8, 0.7]$. Our data elements are executions of workloads in different placements, and the features are some metrics describing the execution.

Model-building methodology and feature selection. To build a model, we use a multi-output *Random Forest* regressor (RF). RF is a machine learning technique known for its ability to learn non-linear functions with very little or no tuning. More complex techniques, like deep neural networks, can yield slightly higher accuracy, but require substantial tuning and are prone to overfitting, especially if not given the "right" features.

Any modelling technique requires predictive input features. Feature selection turned out to be a challenge. In the past, to model performance on multicore systems, researchers used hardware performance events as inputs to the model. In most cases, the HPEs were selected manually[5], which required substantial insight into the intricacies of hardware architecture and its effects on software. Our goal was to make model training automatic, so manual HPE selection was not an option. Automatic selection, on the other hand, turned out to be impractical.

Modern machines have many hundreds of HPEs, some more than 1000 [32]. Automatic feature selection would measure *all* HPEs during training and use feature selection to identify the best predictors. Only four HPEs can be measured at a time, because there is usually only four hardware counter registers, so measuring all the HPEs for the entire training set can take weeks (66 days on our Intel machine), even if we use sampling.

In an effort to find an acceptable compromise, we first used a combination of the manual and automatic approaches. We started with a set of plausible features (41 HPEs on the Intel test system and 25 the AMD) covering cache, memory, TLB, interconnect, and pipeline behaviour, which are metrics commonly used in similar work. We then used *Sequential Forward Selection* [10, 16] (SFS) to pick the best ones. The final RF model would take a vector of selected HPEs observed in a single baseline placement as the input and produce the performance vector as the output. Even after this rigorous feature selection process, we were not happy with the accuracy (see §6).

Finally, we designed a solution that is more robust, requires little training time, and is largely automatic. Instead of relying on HPEs describing various architectural events, we rely on *observations of actual performance in two different configurations from the set of important placements*. Performance can be measured using instructions per cycle (IPC), transactions per second, or any other application-specific metric – the only requirement is that it must be possible to obtain this metric online. Specifying the performance metric for a container is the only manual part of the process. Beyond that, the training process automatically finds the two of the important placements that give the highest accuracy when used as inputs to the model. The final model takes as inputs the performance observations in these two placements and outputs the predicted performance vector. A separate model is trained for each number of vCPUs used in virtual containers.

The downside of this approach is that at runtime we have to run the container in two placements instead of one before obtaining the predictions, but the advantage is that the predictions are more accurate and we do not have to use the time-consuming feature selection process for each new target hardware.

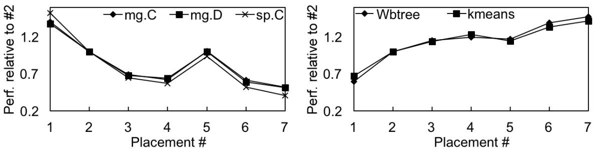


Figure 3: Performance relative to the baseline placement (#2) for workloads in two example clusters on Intel.

Why do performance observations have good predictive ability? Empirical evidence suggests that workloads naturally fall into several categories, according to the shapes of their performance vectors. Figure 3 shows two example categories on the Intel system. As we can see, the vectors within the category are almost identical, but the vectors in different categories are very distinct.

To generate these categories we used k-means clustering. To automatically determine the best value for *k*, we select the *k* that maximizes the average Silhouette coefficient [24, 2] over all data points, which is the standard practice in the field. This clustering method produced six categories on our systems (full results are reported in [13]). This suggests that workloads may naturally form distinct categories depending on their performance trends. For example, workloads that are not memory intensive and are not adversely affected by sharing SMT contexts could belong to the same category (where thread placement does not matter). Another category could be one where using fewer NUMA nodes and fewer physical cores greatly hurts performance, and so on. Then there is no surprise that a ML model could quickly narrow down the category, and hence the shape of the per-

[5] Dwyer's [11] and Zellweger's [32] works are the only exceptions known to us.

formance vector, from two observations of performance.

6 Evaluation

In this section we focus on evaluating the accuracy of predictions. Since our training method does not require automatic feature selection, training the model takes seconds. The algorithms used to determine important placements also run in a matter of seconds. The inference time is negligible (milliseconds).

We had three model variants to compare: the first one used as inputs the actual performance measurements observed in two important placements, the second used only the HPEs observed in a single placement, the third used both. The third variant did not improve accuracy over the first one, so we do not include the data for it.

The set of applications we experimented with are drawn from the NAS Parallel Benchmark suite [6], Parsec suite [7], the Metis map-reduce benchmarks [19], and BLAST [5]. Also included are the Linux kernel compile gcc benchmark, two Spark graph workloads, TPC-C [28] and TPC-H [29] on Postgres and a WiredTiger [3] BTree benchmark. Workloads were run using `lxc` containers and configured to use 16 vCPUs on the AMD system and 24 vCPUs on the Intel system (Fig. 2). Within containers, the number of application threads is set so as to achieve $>$70% CPU utilization on each core, typical of what is done in practice.

Figure 4 show the actual and predicted performance for each workload for important placements on the AMD and Intel systems. The x-axis shows the IDs of the important placements, numbered 1–13 on the AMD system and 1–7 on the Intel system. The y-axis shows the performance in the placements relative to the baseline. Placement #1 was used as the baseline for the AMD system, and placement #2 for the Intel system – the baseline placement can be any of the two placements whose performance is required as the input to the model.

The results are per-application cross-validated. For example, when training the model that will be used for predicting a Spark workload neither the data from *spark-cc* (a Spark connected components algorithm run on the LiveJournal database) nor *spark-pr-lj* (a PageRank algorithm run on the LiveJournal database) is included in the training. We cross-validated every workload, but for space constraints we omit the results for most of the NAS and Parsec benchmarks. They are qualitatively similar to others and we are happy to provide them upon request.

Overall the accuracy when using only the actual performance measurements as model features is high. The predicted performance is within 4.4% of actual on average on the AMD system, and within 6.6% on Intel. A couple of exceptions are the cases where the training set did not include any workloads that behaved similarly to the predicted benchmark, for example *kmeans* on the AMD system, which was the only benchmark in our training set that preferred SMT, or *canneal* on Intel.

Prediction accuracy when using only the HPEs from a single placement was a lot less reliable. On the AMD system it produced good results overall, but the accuracy was still noticeably worse compared to the model variant that relied only on actual performance measurement. On Intel the model relying only on HPEs produced many poor predictions. It completely missed the performance trend for *ft.C* and *freqmine*, produced errors of over 40% for *kmeans* and *WTbtree*, and is noticeably worse for several other workloads.

An example of why HPEs observed in a single placement could have poor predictive power, and one of the reasons why the Intel system produced worse predictions, is predicting the effect of inter-thread communication latency. There is a huge latency difference for communication between a single-node placement and placements including more than one node. For some applications, reduced inter-thread communication latency when all threads are running on a single node has a major performance impact, as is the case for *WTbtree*. Separating the sensitivity to latency from overall memory intensiveness (which can be measured by the cache miss rate) is difficult to do with HPEs. Similarly, it is also very difficult to determine if a workload's working set will fit in a given number of L3 caches by only measuring HPEs on a single placement. ***We conclude that using actual performance observations as model features is likely to produce higher accuracy, in addition to being a more practical method of training the model, than using selected architectural events.***

7 Using the model in practice

There are many ways in which data center operators can use our model. To illustrate one potential use case we set up a scenario, where the user would like to pack as many instances of a given virtual container into a physical server while respecting a performance target. For demonstration of the complete solution we implemented its prototype (covering steps 1-4 in §1). To assess the overheads, we measure the costs of container migration.

We use virtual containers of three types: WiredTiger running a B-tree search workload, Postgres running TPC-H, and Spark running PageRank on a LiveJournal database. For clarity, we present the results of homogeneous configurations, where many containers of the same type are packed into each system. Performance results with our model in heterogeneous configurations can be inferred from these figures, because different containers collocated on the same system do not interfere with our approach. Actual data can be provided upon request.

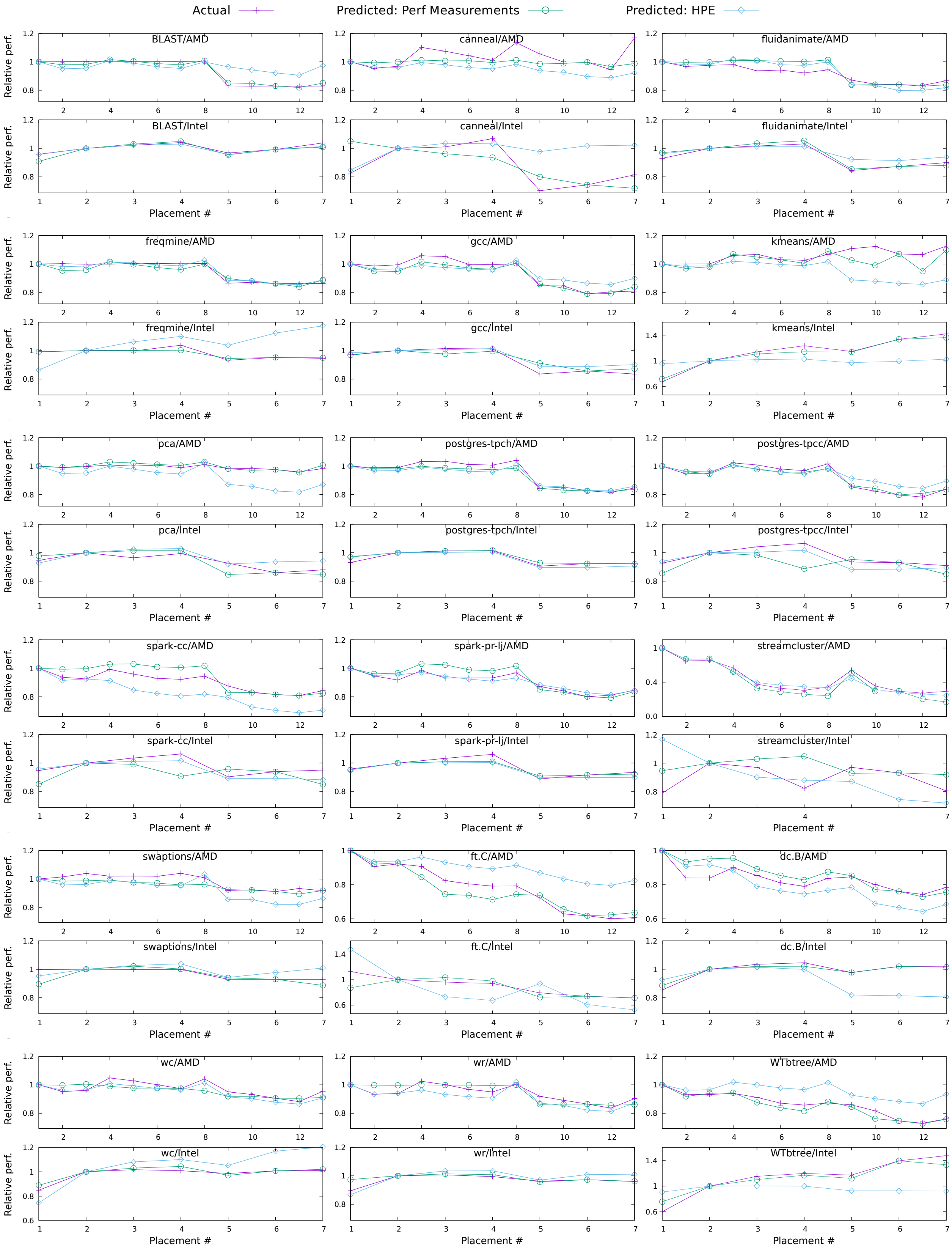
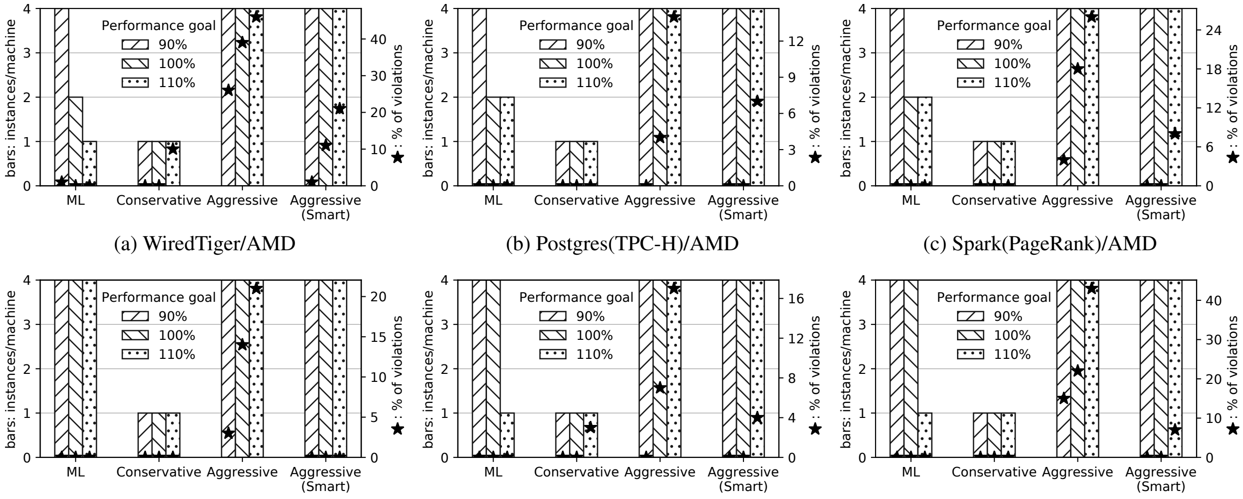


Figure 4: Accuracy of predictions.



(a) WiredTiger/AMD

(b) Postgres(TPC-H)/AMD

(c) Spark(PageRank)/AMD

(d) WiredTiger/Intel

(e) Postgres(TPC-H)/Intel

(f) Spark(PageRank)/Intel

Figure 5: Instances per machine (left y-axis, higher is better) and % performance goal violation (right y-axis, lower is better).

The performance goal can be specified in terms of an application-level metric such as transactions per second or a generic metric such as instructions-per-cycle. The placement policy is agnostic to the metric used and only requires that the application make this metric available at runtime. For simplicity, we set the performance goals to correspond to 90%, 100% and 110% of the performance observed in the baseline placement.

We compare four hypothetical container placement policies. The first policy, referred to as **ML**, is based on our techniques. It decides how many nodes to allocate to the container based on performance observations in two placements and the model presented in the previous section. It runs the workload in two placements during the first few seconds of the execution without interrupting the workload, and then migrates it into the best predicted placement. To separate various aspects of performance, the results shown here do not include the migration overhead; it is studied separately in the next section. The second policy, **Conservative**, is a naïve policy that allocates the entire machine to each instance, allowing only one instance per machine. The third policy, **Aggressive**, is another simple policy that fills the system with as many instances as possible, maximizing machine utilization at the risk of performance violations. For example, our AMD system allows up to four 16-core instances and our Intel system up to four 24-core instances. Neither Conservative nor Aggressive pin vCPUs to cores, allowing Linux to perform the mapping in the way it wishes, and possibly creating unneeded contention. We also evaluate a more sophisticated fourth policy, **Smart-Aggressive**. This policy is similar to Aggressive, except each instance is pinned to the best minimum set of nodes, which we define as having the highest interconnect bandwidth. This policy requires an analysis of the interconnect topology in order to find the correct set of nodes.

We could not make a fair comparison to any other method presented in earlier work. As we explained in §2, most earlier models targeted very different systems and most did not predict performance vectors, so we could not apply them directly.

We evaluate the policies by measuring how many instances of the same workload they were able to pack per machine (higher is better) and the degree of violation of the performance goal as the percent of the target (lower is better). All workloads were run using `lxc` containers and configured to use 16 vCPUs on the AMD system and 24 vCPUs on the Intel system. Figure 5 show the results for the three container types. The bars show the number of instances packed (left y-axis), while the "stars" shows the deviation from the target performance goal, expressed as percentage (right y-axis).

The ML policy always meets the performance goal while in most cases packing more instances per machine than the conservative scheduler. The conservative policy almost always packs fewer instances per machine than ML, but also, surprisingly, may cause performance target violations, because Linux may map vCPUs unevenly to shared resources, causing unnecessary contention.

The aggressive policy packs a maximum possible number of containers per machine, at the cost of performance target violations, up to 46% with WiredTiger on AMD, and 43% with Spark on Intel. It is surprising that even when the aggressive policy packs the same number of containers per machine as the model-based policy, it still often reports a higher violation percent. That is because this policy allows virtual containers to share NUMA nodes. Smart-aggressive addresses this shortcoming, but even that policy can cause performance violations (e.g., 20% for WiredTiger on AMD), because it does not take into account all ways in which workload placement might affect performance.

Memory migration overhead. Memory migration in Linux is known to be inefficient [18]. Since we need to measure the performance of workloads in two or three configurations, fast memory migration is needed in that phase to reduce overheads. Lepers et al. [18] propose a method that freezes the application and migrates pages with concurrent worker threads. We improve on this by migrating the page cache and reducing locking overhead. Table 2 shows migration times for the workloads of §5. We observed similar results on the Intel system. Note that page cache migration time is counted with our method only since Default Linux doesn't support it – and yet, it can be a large part of migration overhead (93% with BLAST, 75% with TPC-C and 62% on TPC-H). We are able to migrate a large amount of memory in a few seconds, usually one order of magnitude faster than Default Linux (38× faster for Spark). Linux is especially inefficient for workloads with many processes such as TPC-C, since it has per-task overhead linked to updating the `cpuset` at each migration.

A drawback of our method is that it requires freezing the container during migration in order to reduce contention on some critical kernel locks. It is therefore suitable for non-latency-sensitive workloads. For latency-sensitive workloads, we have the option of not freezing the container and to instead throttle the bandwidth given to the migration process so as to reduce the impact on the running application. Thus, the migration takes more time but with a smaller impact on the running container. Using this method, the overhead of migration for the WiredTiger workload[6] is between 3% and 6%, and the migration takes 60 seconds. In comparison, Linux takes 43.8 seconds, has a overhead of 20% at best and completely freezes the applications for several seconds. It also does not migrate the page cache.

Overall, we observe that the migration overhead is proportional to the amount of memory used by the container, except in cases with extremely high thread counts. Using the container's memory footprint, the user can estimate whether the migration cost warrants an online deployment of the placement algorithm, or if it is preferable to use it offline for placement of recurring jobs.

Benchmark	Memory (GB)	Fast Migration (s)	Default Linux (s)
BLAST	18.5	3.0	5.9
canneal	1.1	0.3	3.9
fluidanimate	0.7	0.3	2.3
freqmine	1.3	0.3	4.2
gcc	1.4	0.3	2.8
kmeans	7.2	1.5	6.5
pca	12.0	2.8	10.0
postgres-tpch	26.8	5.8	117.1
postgres-tpcc	37.7	14.9	431.0
spark-cc	17.0	3.7	139.9
spark-pr-lj	17.1	3.8	137.0
streamcluster	0.1	0.1	0.4
swaptions	0.01	0.1	0.0
ft.C	5.0	1.3	19.4
dc.B	27.3	5.4	51.7
wc	15.4	3.4	19.5
wr	17.1	3.6	18.9
WTbtree	36.3	6.3	43.8

Table 2: Migration performance on the AMD system, compared to the default Linux migration method. The amount of memory includes processes' memory and the page cache associated with the container.

8 Conclusion

Modern multicore systems have a complex hierarchy of shared resources and performance can vary wildly depending on how virtual CPUs are mapped to hardware contexts. Operators waste resources and money by using conservative and sub-optimal placement policies. We have shown a solution to this problem using a methodology to abstract a system's shared resources, identify important placements, and predict their performance. Our method can lead to very significant advantages in machine utilization while keeping performance guarantees.

CPU architecture is continually changing, often by sharing resources between cores in new ways, in order to continue scaling the core count. AMD's newly introduced Zen architecture [8] has L3 cache sharing separate from sharing the memory controller. Intel's Haswell-E architecture has asymmetric links between NUMA nodes through its cluster-on-die feature [22], which has unique performance implications different from other asymmetric architectures. The flexibility of our methods means that they can be used on systems like these or future architectures without significant retooling by an expert.

References

[1] Amazon EC2 Instance Types. `https://aws.amazon.com/`

[6] We picked WiredTiger for this evaluation since other the other workloads we use don't report the evolution of performance during the execution.

ec2/instance-types.

[2] Selecting the number of clusters with silhouette analysis on KMeans clustering. http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html.

[3] The WiredTiger key-value store. http://www.wiredtiger.com.

[4] Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 469–482.

[5] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. Basic local alignment search tool. *Journal of molecular biology 215*, 3 (1990), 403–410.

[6] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. The Nas Parallel Benchmarks. *IJHPCA 5*, 3 (1991), 63–73.

[7] Bienia, C., and Li, K. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation* (2009).

[8] Clark, M. A New, High Performance x86 Core Design from AMD. In *Hot Chips: A Symposium on High Performance Chips* (2016).

[9] Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., and Roth, M. Traffic management: A holistic approach to memory placement on numa systems. *SIGPLAN Not. 48*, 4 (Mar. 2013), 381–394.

[10] Draper, N. R., and Smith, H. *Applied regression analysis*. John Wiley & Sons, 1966.

[11] Dwyer, T., Fedorova, A., Blagodurov, S., Roth, M., Gaud, F., and Pei, J. A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 83:1–83:11.

[12] Fedorova, A., Seltzer, M., and Smith, M. D. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques* (Washington, DC, USA, 2007), PACT '07, IEEE Computer Society, pp. 25–38.

[13] Funston, J. R. *A model for thread and memory placement on NUMA systems*. PhD Dissertation, University of British Columbia https://open.library.ubc.ca/cIRcle/collections/ubctheses/24/items/1.0363032, 2018.

[14] Funston, J. R., El Maghraoui, K., Jann, J., Pattnaik, P., and Fedorova, A. An smt-selection metric to improve multithreaded applications' performance. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2012), IPDPS '12, IEEE Computer Society, pp. 1388–1399.

[15] Goodman, D., Varisteas, G., and Harris, T. Pandia: Comprehensive contention-sensitive thread placement. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 254–269.

[16] John, G. H., Kohavi, R., and Pfleger, K. Irrelevant features and the subset selection problem. In *Machine learning: proceedings of the eleventh international conference* (1994), pp. 121–129.

[17] Knauerhase, R., Brett, P., Hohlt, B., Li, T., and Hahn, S. Using os observations to improve performance in multicore systems. *IEEE Micro 28*, 3 (May 2008), 54–66.

[18] Lepers, B., Quéma, V., and Fedorova, A. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2015), USENIX ATC '15, USENIX Association, pp. 277–289.

[19] Mao, Y., Morris, R., and Kaashoek, F. Optimizing MapReduce for multicore architectures. Tech. rep., 2010.

[20] McCalpin, J. D. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[21] Merkel, A., Stoess, J., and Bellosa, F. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 153–166.

[22] Molka, D., Hackenberg, D., Schöne, R., and Nagel, W. E. Cache coherence protocol and memory performance of the intel haswell-ep architecture. In *Parallel Processing (ICPP), 2015 44th International Conference on* (2015), IEEE, pp. 739–748.

[23] Radojkovic, P., Carpenter, P. M., Moretó, M., Cakarevic, V., Verdú, J., Pajuelo, A., Cazorla, F. J., Nemirovsky, M., and Valero, M. Thread assignment in multicore/multithreaded processors: A statistical approach. *IEEE Trans. Computers 65*, 1 (2016), 256–269.

[24] Rousseeuw, P. J. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics 20* (1987), 53–65.

[25] Snavely, A., and Tullsen, D. M. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2000), ASPLOS IX, ACM, pp. 234–244.

[26] Sridharan, S., Gupta, G., and Sohi, G. S. Adaptive, efficient, parallel execution of parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 169–180.

[27] Tam, D., Azimi, R., and Stumm, M. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 47–58.

[28] Transaction Processing Performance Council. TPC Benchmark C. http://www.tpc.org/tpcc/, 2010.

[29] Transaction Processing Performance Council. TPC Benchmark H. http://www.tpc.org/tpch/, 2014.

[30] Yadwadkar, N. J., Hariharan, B., Gonzalez, J. E., Smith, B., and Katz, R. H. Selecting the *best* VM across multiple public clouds: a data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017* (2017), pp. 452–465.

[31] Yang, H., Breslow, A., Mars, J., and Tang, L. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 607–618.

[32] Zellweger, G., Lin, D., and Roscoe, T. So many performance events, so little time. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems* (New York, NY, USA, 2016), APSys '16, ACM, pp. 14:1–14:9.

[33] Zhang, Y., Laurenzano, M. A., Mars, J., and Tang, L. Smite: Precise QoS prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), IEEE Computer Society, pp. 406–418.

[34] Zhuravlev, S., Blagodurov, S., and Fedorova, A. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS XV, ACM, pp. 129–142.

Getting to the Root of Concurrent Binary Search Tree Performance

Maya Arbel-Raviv
Technion

Trevor Brown
IST Austria

Adam Morrison
Tel Aviv University

Abstract

Many systems rely on optimistic concurrent search trees for multi-core scalability. In principle, optimistic trees have a simple performance story: searches are read-only and so run in parallel, with writes to shared memory occurring only when modifying the data structure. However, this paper shows that in practice, obtaining the full performance benefits of optimistic search trees is not so simple.

We focus on optimistic binary search trees (BSTs) and perform a detailed performance analysis of 10 state-of-the-art BSTs on large scale x86-64 hardware, using both microbenchmarks and an in-memory database system. We find and explain significant unexpected performance differences between BSTs with similar tree structure and search implementations, which we trace to subtle performance-degrading interactions of BSTs with systems software and hardware subsystems. We further derive a prescriptive approach to avoid this performance degradation, as well as algorithmic insights on optimistic BST design. Our work underlines the gap between the theory and practice of multi-core performance, and calls for further research to help bridge this gap.

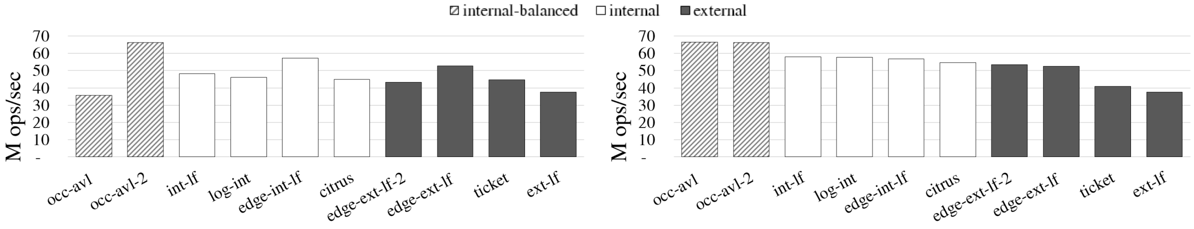
1 Introduction

Many systems rely on optimistic concurrent search trees for multi-core scalability. (For example, in-memory databases [35], key/value stores [29], and OS virtual memory subsystems [10].) Optimistic search trees seem to have a simple performance story, based on the observation that to scale well a workload must contain sufficient high-level parallelism (e.g., operations should not all modify the same key [21]). Optimistic search trees therefore strive to avoid synchronization contention between operations that do not conflict semantically, such as updates to different keys. In particular, optimistic trees use *read-only* searches, which do not lock or otherwise write to traversed nodes, with writes to shared memory occurring only to modify the data structure [7, 29]. This design is considered key to search tree performance [12, 18].

We show, however, that realizing the full performance benefits of optimistic tree designs is far from simple, because their performance is affected by subtle interactions with systems software and hardware subsystems that are hard to identify and solve. To demonstrate this issue, consider the problem faced by systems designers who need to reason about data structure performance. Given that real-life search tree workloads operate on trees with millions of items and do not suffer from high contention [3, 26, 35], it is natural to assume that search performance will be a dominating factor. (After all, most of the time will be spent searching the tree, with synchronization—if any—happening only at the end of a search.) In particular, we would expect two trees with similar structure (and thus similar-length search paths), such as balanced trees with logarithmic height, to perform similarly.

In practice, however, this expectation turns out to be false. We test the above reasoning on optimistic *binary* search trees (BSTs), since there are BST designs with various tree structures [2, 7, 14, 15, 22, 23, 32, 33]. We find significant performance differences between BSTs with similar structure and traversal techniques. Figure 1a depicts examples of such anomalies. (We show a *read-only* workload, consisting only of lookups, to rule out synchronization as a cause. We detail the studied BSTs and experimental setup in § 2.) For instance, one unbalanced internal BST (*edge-int-lf*) outperforms other BSTs with the same tree structure (*log-int* and *citrus*). There is even a significant difference between two implementations of the same BST algorithm (*occ-avl* and *occ-avl-2*).

The goal of this work is to explain and solve such unexpected performance results. We perform a detailed performance analysis of 10 state-of-the-art optimistic BST implementations on large scale x86-64 hardware, in which we uncover the root causes of the observed anomalies. Using microbenchmarks, we find that performance anomalies are caused by multiple performance-degrading interactions of BSTs with systems software and hardware subsystems, mostly related to cache effects. These cache effects are due either to cache-unfriendly implementation oversights or, more interestingly, to memory layout pathologies that are caused by interactions between the BST and the memory allocator. To determine whether our observations are only artifacts of micro benchmarking, or whether similar issues appear in more complex software, we deploy the BSTs as the index structure in DBx1000, an in-memory database [4, 27, 39, 40]. We find that similar anomalies exist in DBx1000 as well. Most importantly, we find that a simple approach of *segregating* BST-related allocations, so that BST data is not mixed with application data, improves performance of the BSTs by up to 20%



a. Current state: Multiple anomalies.

b. Following our analysis and fixes.

Figure 1: Unexpected BST performance results: Aggregated throughput (BST operations/second) of a 64-thread read-only (100% `lookup`) workload on 1 M-item tree executed on a 64-core AMD machine.

and of the overall application by up to 10%. Figure 1b demonstrates part of our results.

Our work underlines the gap between the theory and practice of multi-core performance. As we show, it is non-trivial to understand a search tree's performance, and specifically, whether performance is due to fundamental algorithmic factors or to implementation issues. While we focus on BSTs, the effects we uncover are relevant to other optimistic concurrent data structures, as they stem from general principles of memory allocator and systems design. (But we leave such analysis for future work.) Our results therefore call for further research to help bridge the gap between the principles and practice of multi-core performance, to simplify the task of deploying a concurrent data structure and reasoning about its performance.

2 Scope

2.1 BSTs

We analyze C implementations of 8 BST algorithms, two of which have independent implementations, for a total of 10 implementations. The algorithms implement the standard key/value-map operations, `lookup`, `insert` and `remove`. [1] Table 1 lists the implementations studied. These BSTs span the known points in the design space, covering combinations of synchronization techniques, tree types (internal vs. external), and balancing choices (unbalanced vs. self-balancing algorithms).

All BSTs but *int-lf* feature read-only traversals; in *int-lf*, a traversal might synchronize with a concurrent update. In the *lock-free* BSTs, updates manipulate the data structure using atomic instructions, such as `compare-and-swap` (CAS), instead of synchronizing with locks. Both *int-lf* and *ext-lf* use *operation descriptor* objects to implement *helping* between their operations. A descriptor details the memory modifications that an update operation needs to perform. Before performing its modifications, the update operation CASes a pointer to its descriptor in each of the nodes it needs to update. Other operations that encounter the descriptor use the information therein to help the update complete. *edge-int-lf* and *edge-ext-lf* avoid descriptors by "stealing" some bits from node left/right pointers to encode helping-related information.

name	**synchronization technique**	**tree type**	**self-balance?**	**impl. source**†
occ-avl [7]	fine-grained locks	part. ext	✓	[22]
occ-avl-2			✓	ASCYLIB
edge-int-lf [33]	lock-free	int		authors
log-int [14]	fine-grained locks	int		ASCYLIB
citrus [2]	fine-grained locks	int		authors
int-lf [23]	lock-free	int		ASCYLIB
edge-ext-lf [32]	lock-free	ext		authors
edge-ext-lf-2				ASCYLIB
ticket [12]	fine-grained locks	ext		authors
ext-lf [15]	lock-free	ext		ASCYLIB

† *authors* refers to original authors' implementation, and *ASCYLIB* to the implementation in the ASCYLIB library [12].

Table 1: BST implementations studied (ordered by the expected performance of searches).

An *internal* BST stores an item in every node, whereas an *external* BST stores items only in leaves. Internal BSTs have different solutions for removing a node with two children while maintaining consistency of concurrent searches. *edge-int-lf* and *int-lf* searches use *validation* to detect such a concurrent removal and restart the search. *log-int* avoids restarts by having an unsuccessful search (i.e., that fails to find its target key) traverse an ordered list which links all nodes, to verify that the key is indeed not present. Finally, *occ-avl* marks a node with two children as logically removed instead of physically removing it from the data structure, resulting in a *partially external* tree. *occ-avl* uses validation to restart a search that could take the wrong path due to a concurrent tree rotation.

The BSTs appear in Table 1 according to their expected relative performance in workloads where search time dominates performance: all else being equal, one expects self-balancing BSTs, which maintain logarithmic height, to outperform unbalanced BSTs; and internal BSTs to outperform external BSTs.

The original implementation of two of the BSTs [7, 14] is in Java. We choose, however, to evaluate 3rd-party C implementations of these BSTs, to obtain an apples-to-apples comparison and to simplify the analysis.

We fixed incorrect use of the C `volatile` keyword in some of the evaluated implementations. In general, to

[1] Two implementations [32, 33] originally implemented set semantics, storing only keys, but we modify them to hold values as well.

system name	*abu-dhabi*	*haswell*
processors	4× AMD Opteron 6376 (Abu Dhabi)	2× Intel Xeon E7-4830 v3 (Haswell)
# cores/proc	16 (2 dies w/ 4 modules, 2 cores per module)	12 (24 hyperthreads)
core freq.	2.3 GHz	2.1 GHz
L1d cache	16 KiB, 4-way	32 KiB, 8-way
L2d cache	2 MiB, 16-way (per mod.)	256 KiB, 8-way
last-level cache (LLC)	2 × 8 MiB, 64-way (shared, per die)	30 MiB, 20-way (shared)
interconnect	6.4 GT/s HyperTransport (HT) 3.0	6.4 GT/s QuickPath Interconnect (QPI)
memory	128 GiB Sync DDR3-1600 MHz	128 GiB Sync DDR3-1600 MHz

Table 2: Hardware platforms.

name	bloated nodes	scattered fields	heavy traversals
occ-avl	✓	✓	
occ-avl-2		~~✓~~†	
edge-int-lf			✓
log-int	✓	✓	
citrus	✓		
int-lf	✓		
edge-ext-lf			
edge-ext-lf-2	✓		
ticket	✓		
ext-lf	✓		✓

† *occ-avl-2* has a search field not at the start of the node, but this does not cause extra cache misses, as the nodes are cache line-sized.

Table 3: BST implementation issues.

avoid such problems, one should either use C `atomics`, or place the `volatile` keyword correctly: a volatile pointer to a node is written `node * volatile ptr`, not `volatile node * ptr`.

2.2 Experimental setup

We perform experiments on two multi-socket x86 platforms, by AMD and Intel. Table 2 details the hardware characteristics of these platforms. Both machines are NUMA platforms, configured so that DRAM is equally divided between the NUMA nodes. When running experiments, we use the standard practice of interleaving the benchmark's memory pages across the system's NUMA nodes (using the `numactl` command) to prevent any NUMA node from becoming a bottleneck. We compile the benchmarks with `gcc` v4.8. As in prior work, we use a scalable memory allocator (`jemalloc` [16]) to prevent memory allocation from becoming a bottleneck.

3 BST performance in isolation

We begin by analyzing BST performance on the standard microbenchmark used in the concurrency literature [2, 7, 11, 14, 22, 23, 32, 33], which models an application using a BST. The benchmark consists of a loop in which each thread repeatedly performs a random BST operation on a random integer key, and its performance metric is the obtained aggregate throughput of BST operations. We find that several implementations make simple oversights that lead to inefficient BST searches, but that fixing these problems still leaves many unexpected results (§ 3.1). These remaining anomalies occur due to cache behaviour differences due to BST memory layout (§ 3.2) and due to subtle interactions with the prefetching units (§ 3.3).

3.1 BST implementation issues

Most of the BST implementations contain one or more of three implementation oversights that negatively impact the performance of BST searches. Table 3 summarizes our findings, which we discuss next:

Bloated nodes Most implementations unnecessarily bloat the tree nodes, reducing the amount of the tree that can fit in each level of the cache hierarchy. Some lock-based implementations use `pthread` mutex locks, which occupy 40 bytes, instead of `pthread` spin locks, which occupy 4 bytes. Several implementations pad BST nodes to cache line size, presumably to avoid false sharing.

Scattered fields Fields commonly read by traversals (key/left/right, as well as fields related to detecting concurrent tree modifications) should be located first in the node structure, to minimizes the chance that a search accesses two cache lines when traversing a node.

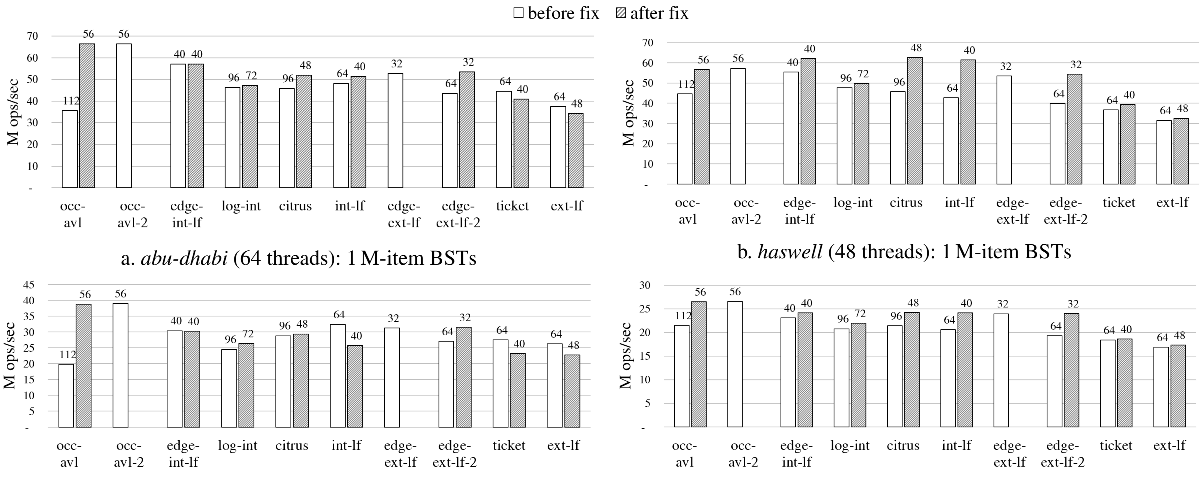
Heavy traversals *edge-int-lf* and *ext-lf* base all operations on one shared traversal method, and so end up burdening `lookup` operations with the book-keeping required only for updates, such as maintaining pointers to the parent/grandparent of the current node.

3.1.1 Evaluating impact of implementation issues

We fix the above implementation issues by replacing `pthread` mutex locks with spin locks, removing padding and reordering node fields in the affected implementations, and evaluate the impact of these fixes.

Methodology Our benchmark is parameterized by the distributions that the operation types and keys are chosen from, the size of the key space, and the number of items (key/value pairs) initially present in the tree. Following the practice in the concurrency research literature, we (1) choose operation keys uniformly at random; (2) perform `insert` and `remove` with equal probability throughout the benchmark; and (3) initialize the BSTs (using concurrent `insert()`s) with $U/2$ random items, where U is the size of the key space. We report averages of five 3-second runs on an otherwise idle system.

Results Figure 2 shows the performance impact of our changes on trees that initially contain 1 M and 10 M items, to model realistic working sets. We show results from read-only (100% `lookup`) workloads so that we can reason about search performance and remove synchronization effects as a confounding factor. We have, however, verified that read-only workloads are a good proxy for read-dominated workloads on this benchmark: e.g., in workloads with 90% `lookups`, the relative performance order of the BSTs matches that of the read-only case almost perfectly and most comparison points remain similar



a. *abu-dhabi* (64 threads): 1 M-item BSTs

b. *haswell* (48 threads): 1 M-item BSTs

c. *abu-dhabi* (64 threads): 10 M-item BSTs

d. *haswell* (48 threads): 10 M-item BSTs

Figure 2: Impact of fixing BST implementation issues. Numbers on top of the bars show the BST node size.

even at a 70% lookup rate.

We show results from executions with the maximum number of threads on each platform, as all BSTs scale with the amount of concurrency. On the 1 M-item tree, our fixes improve the throughput of the BSTs by up to 86% on *abu-dhabi* and by up to 43% on *haswell*, with a geo mean improvement of 11% on *abu-dhabi* and 23% on *haswell*. Moreover, reducing *occ-avl*'s node size brings its performance to the level of *occ-avl-2*.

Unexpected results Several unexpected results remain even after fixing the BST implementation issues, and we uncover their cause in the remainder of this section:

- Why does decreasing node size *hurt* throughput for 10 M-item *int-lf*, *ticket* and *ext-lf* on *abu-dhabi*? (§ 3.2.2)
- Why does *int-lf* benefit from a reduction of 64-byte nodes to 48-byte nodes much more than *ticket* on *haswell*? Why does *log-int* perform worse than the other unbalanced internal BSTs? (§ 3.2.3)
- Why does *edge-ext-lf* outperform other external BSTs, when they all have the same tree structure? (§ 3.2.4)
- Why do *occ-avl* and *occ-avl-2*, self-balancing BSTs, behave differently on *abu-dhabi* and *haswell*? On *abu-dhabi* they significantly outperform unbalanced trees (as expected), whereas on *haswell* they do not. (§ 3.3)

3.2 Memory layout issues

We trace most of the anomalies to memory layout issues that lead to different cache behaviours between the BSTs. These memory layout issues result from subtle interactions between the BST's allocation pattern and the policies of the memory allocator, particularly the use of *segregated free lists* [24] for satisfying allocations.

3.2.1 Segregated free list allocation

At a high level, scalable memory allocators [5, 16, 17] avoid contention by providing each thread with its own heap. These heaps are implemented as a set of free lists [24], one for each possible *size class*. Free lists are generally implemented as *superblocks*, which contain an array of blocks. To satisfy an *n*-byte allocation request, the allocator rounds *n* up to the nearest size class, *s*, and returns an *s*-byte block obtained from the relevant free list. In the jemalloc memory allocator we used for our experiments, the size classes used for allocations of up to 1 KiB are $8, 16, 32, 48, 64, 80, 96, 112, 128, 192, 256, 320, 384, 448,$ 512, and 1024. In addition to size classes, allocators differ in the structure and size of superblocks, the algorithm for mapping a block to its superblock, policies for allocating and releasing superblocks, and synchronization schemes. The important point in our context is that we can model the behaviour of the memory allocator as satisfying allocations of size *s* from an array of blocks of size *s*.

3.2.2 Crossing cache lines

In the BSTs we study, visiting a node should in principle incur at most one cache miss: the size of the searched fields (key, child pointer, and any fields used to synchronize with concurrent updates) fit in one cache line. We find, however, that the memory allocator might place a node in memory so that these search fields straddle a cache line boundary, causing a visit to the node to incur 2 cache misses.

Consider, for example, a BST whose searches access the first 24 bytes of a node (8-byte key and 8-byte left or right child pointer). If its node size is 48 bytes and the memory allocator's block array is cache line-aligned, then nodes will start at offsets 0, 16, 32, and 48 within cache lines. For the nodes at offset 48, the last 8 bytes of these searched fields extend into the next line, possibly leading to a cache miss. Such a miss occurs with probability $1/8$, as one in 4 nodes straddles a cache line boundary, and a search reads each next pointer with probability $1/2$.

Originally, *int-lf*, *ticket* and *ext-lf* do not experience such cache line crosses, as they have padded 64-byte nodes that the memory allocator allocates from a size class of 64-byte blocks. Decreasing their node size introduces this issue, whose performance impact is a trade-off that depends on the workload. At one end of the spectrum, if a smaller node size allows the entire tree to fit into the LLC, then one can eliminate cache misses altogether. At the other end, if the workload is such that almost every node traversed incurs a cache miss, then it is better to increase the node size to avoid crossing cache lines, as otherwise the expected number of cache misses per search increases by the expected number of nodes whose search incurs an extra miss (e.g., by $1\frac{1}{8}\times$ for 48-byte nodes).

In our workloads, we observe a 17% (geo mean) throughput degradation on the 10 M-item tree on *abudhabi*, but negligible overhead on the 1 M-item tree. We do not observe this anomaly on *haswell* because it has an adjacent-line prefetcher [36] that effectively doubles the cache line size and hides the effect of misses caused by cache line crossings.

3.2.3 Underutilized caches due to allocation pattern

We find that BST allocation patterns can lead to *cache set underutilization*,[2] in which the workload uses some cache sets more than others, thereby leading to increased associativity misses on the overused cache sets. We identify two causes for underutilized cache sets. First, the memory allocator might place allocated nodes in memory so that they map to just a subset of the cache sets. More insidiously, even if the nodes cover all cache sets but are allocated next to cache lines containing useless data, then prefetching this data evicts useful nodes from the cache.

We demonstrate cache set underutilization that occurs in *int-lf*; *log-int* has a similar issue, whose description we omit due to space constraints.

***int-lf* analysis** We observe an anomaly on *haswell*, in which *int-lf* benefits from a reduction of 64-byte nodes to 48-byte nodes much more than *ticket*. We focus on the 1 M-item tree experiment. While performance counter data shows that *int-lf*'s throughput improvement with smaller nodes is correlated with reduced LLC miss rates, the 1 M-item tree should almost fit into *haswell*'s 30 MiB LLC even with bloated nodes. This points to a cache set underutilization problem, in which *int-lf* effectively runs as if with a smaller cache. We verify this hypothesis by computing the cache set indexes of each node,[3] finding that the original *int-lf* implementation uses only 50% of the L2 and LLC sets. Next, we analyze *int-lf*'s allocation pattern to find the cause for this problem.

int-lf variant	ops/sec	unused L1 sets	unused L2 sets	unused L3 sets
64 b node	42.5M	1.6%	50.8%	50.8%
64 b node, w/ allocs	42.5M	1.6%	1.6%	1.6%
64 b node, w/ allocs, no prefetching	60.0M	1.6%	1.6%	1.6%
40 b node	60.0M	1.6%	1.6%	1.6%

Table 4: *int-lf* on *haswell* cache set usage (1 M-item BST).

Like many lock-free algorithms, *int-lf* uses *operation descriptors* so that threads can help each other to complete their operations (see § 2.1). Each thread's allocation pattern during the BST initialization is thus *NDNDND*..., as each `insert` operation allocates a new node of size N and a descriptor of size D. The size of both descriptors and *int-lf*'s original padded nodes is 64 bytes, the cache line size. Both allocation types are thus satisfied from the same allocator size class, and consequently, nodes occupy only even (or only odd) cache set indexes, utilizing only 50% of the available cache sets. (We note that *int-lf* intentionally does not free descriptors, to avoid an ABA problem[4] on the descriptor-pointer field in the nodes. The idea is that if the content of this field only changes from one descriptor to another, an ABA problem cannot occur.)

Fixing cache set underutilization Shrinking *int-lf*'s node size as part of fixing its implementation oversights has the serendipitous effect of *segregating* node and descriptor allocations. As nodes and descriptor allocations become satisfied from different size classes, nodes occupy all cache sets. Moreover, only nodes are allocated from their size class, and so no prefetching of useless data occurs. To prevent cache set underutilization in a principled way, we explicitly segregate BST nodes by allocating them from a dedicated memory pool; see § 4 for details.

It remains to show that cache set underutilization is not only caused by mapping nodes to a strict subset of the cache. To this end, we modify the microbenchmark to add allocation calls of random sizes between BST operations. These random allocations break the benchmark's regular allocation pattern, causing *int-lf* nodes to map to all cache sets. Nevertheless, unless we additionally disable prefetching,[5] *int-lf* performs poorly. Table 4 shows the result of our experiments. Fixing cache set underutilization improves throughput by 40% on the 1 M-item tree.

3.2.4 Collocated children

We find that the high throughput obtained by *edge-ext-lf* compared to the other external BSTs is due to a fortunate allocation pattern, which causes many leaves to be

[2] An 2^n-way associative cache of size 2^C bytes with 2^l cache lines groups its slots into *sets* of size 2^{C-l-n}. Bits $l+1,\ldots,C-n+1$ of an address determine its set index.

[3] We compute LLC set indexes using the physical addresses of the nodes. Specifically, we use the techniques of [30, 38] to reverse engineer the mapping from physical address to *haswell* LLC cache slices.

[4] An ABA problem occurs when a thread reads the same value (A) from a location twice, interpreting this to mean that the location has contained (A) at all times between the two reads, whereas between the two reads, the location was actually changed to (B) and back to (A).

[5] Specifically, the L1 data cache prefetcher.

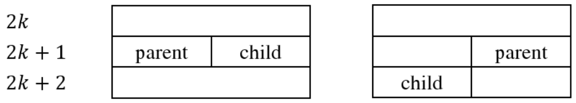


Figure 3: Collocated and shifted nodes in *edge-ext-lf*.

collocated on the same cache line with their parent.

edge-ext-lf is an external BST with immutable 32-byte nodes: an `insert` whose search completes at leaf u allocates a new internal (routing) node v and a new leaf node (with the inserted item) w, which is a child of v. It then replaces u with v. The memory allocator satisfies node allocations from a superblock of 32-byte blocks. Therefore, v and w might be collocated on the same cache line.

We analyze the node addresses in the evaluated trees and find that 75% of the internal nodes which have a leaf child are also collocated in the same cache line with one of their children. (This collocation can only occur for leaves. Whenever *edge-ext-lf* extends a path, it breaks the previous parent/child collocation.)

To evaluate the performance impact of the collocation property, we implement *shifted* versions of *edge-ext-lf*, where we add one 32-byte allocation before the initialization of the tree. This shifts the cache line offsets of all later allocations, moving the child nodes to a different cache line (Figure 3). As expected, we find that in the shifted implementations, 75% of internal nodes which have a leaf child have a child located on the adjacent cache line. On a 1 M-item BST, we observe throughput slowdowns of 14% and 11% on *abu-dhabi* and *haswell*, respectively.

The reason that prefetching does not hide this problem is again due to the allocation pattern. We examine the node addresses and find that 100% of the nodes which have a leaf child in the next cache line are themselves located on an odd cache line (Figure 3). The adjacent-line prefetcher on *haswell* "fetches the cache line that comprises a cache line pair" [36]. This appears to imply that it is only triggered on accesses to an even cache line, and thus is ineffective in this case.

3.3 Prefetching issues

Bronson et al.'s relaxed balance AVL BST [7] (*occ-avl* and *occ-avl-2*) is considered as one the fastest BSTs. While on *abu-dhabi* the AVL tree indeed outperforms the other BSTs by a geo mean of 40% in both tree sizes, on *haswell* it is not the best performer on the 1 M-item tree experiment. We trace this anomaly to a novel interaction of the BST's optimistic concurrency control (OCC) and the L2 prefetcher, which is exposed after removing a different bottleneck in the OCC implementation.

The algorithm uses *versioning*—an OCC implementation technique—to detect concurrent tree modifications during searches. Glossing over some details, each child pointer has an associated version number that increases when the pointer is updated. Observing that this version has not changed between time t_0 and t_1 allows a search to verify that the associated pointer has not changed as well. Searches use this property to verify that they traverse through a node only if both the inbound pointer to u and the outbound pointer to the next node on the path were valid together at the same point in time.

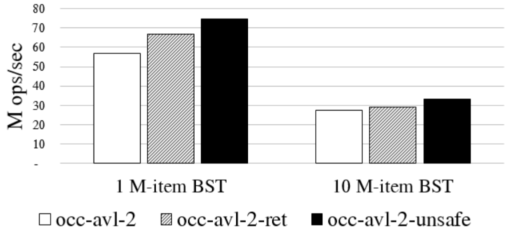


Figure 4: Impact of OCC changes in *occ-avl-2* (*haswell*).

When the validation at some node u fails, the search starts ascending along the traversed path, revalidating at each node, until it returns to a consistent state from which it resumes the search. Both *occ-avl* and *occ-avl-2* use recursive calls to visit nodes, thereby recording this bookkeeping data on the stack. This information, however, is used only if a search encounters a concurrent update, which is expected to be a rare event. We therefore change *occ-avl-2* to restart the search from the root when validation fails, yielding the *occ-avl-2-ret* implementation.

In the 1 M-item tree experiment, *occ-avl-2-ret* outperforms *occ-avl* and *occ-avl-2* by 17% on *haswell*. We observe, however, that it generates many L2 prefetch misses. Our workload does not benefit from hardware prefetching, since the next cache line a BST search visits is random. Prefetching thus hurts BST throughput, as it evicts potentially useful tree nodes (e.g., nodes at the top of the tree) from the cache.

We find that reading the version stored in the nodes triggers an L2 prefetch. While reading twice from the cache line (key and next pointer) does not trigger prefetching, *any* additional read from the node does. To evaluate the impact of prefetching, we implement a variant of the algorithm without the version reads, *occ-avl-2-unsafe*. This variant is safe to run only in a read-only workload; we use it just to estimate the performance lost due to prefetching. Figure 4 shows the results: On a 1 M-item tree, *occ-avl-2-unsafe* improves a further 12% over *occ-avl-2-ret*, for an overall 31% improvement over *occ-avl-2*; its total improvement over *occ-avl-2* on a 10 M-item BST is 21%.

4 BST performance in an application

The next logical step is to ask whether these performance issues are simply artifacts of micro benchmarking, or whether similar issues appear in more complex software. To this end, we study an *in-memory database management system* called DBx1000 [39] (henceforth, simply DBx), which is used in multi-core database research [4, 27, 39, 40]. In this section, we focus on the *haswell* machine.

DBx DBx implements a single relational database, which contains one or more *tables*, each of which consists of a sequence of *rows*. It offers a variety of different concurrency control mechanisms for allowing processes to access tables and rows. We use its 2-phased locking option, which locks individual rows of tables, and has been shown to scale on simulated systems containing up to one thousand cores [39].

Each table can have one or more *key fields* and associated *indexes*. Each index allows processes to query a specific key field, quickly locating any rows in which the key field contains a desired value. Any thread-safe data structure can serve as an index in DBx, as long as it implements a *multimap*. A multimap represents a set of keys, each of which maps to one or more *values* (pointers to rows of a table), and offers three operations: `search`(key), `insert`$(key, value)$ and `remove`$(key, value)$. `search`(key) returns all of the values to which *key* maps in the multimap. `insert`$(key, value)$ adds a mapping from *key* to *value*. If *key* maps to *value*, then `remove`$(key, value)$ removes the mapping from *key* to *value*, and returns true. Otherwise, it simply returns false.

Methodology We replace the default index implementation in DBx with each of the BSTs that we study. To do so, we had to overcome a minor complication: each of these BSTs implements a *map*, not a *multimap*. That is, each *key* maps only to a single value. We transformed these maps into multimaps as follows. Instead of storing keys and pointers to rows in the map, each key maps to the head of a linked list that is protected by a lock. (The locks are stored in a separate lock table.) Then, to perform `insert`$(key, value)$ on the multimap, where *value* is a pointer to a *row*, we simply insert *row* into the appropriate linked list in the underlying map.

Workloads To analyze the performance of the various BST implementations in DBx, we use the well known Yahoo! Cloud Serving Benchmark (YCSB), and the Transaction Processing Performance Council's TPC-C benchmark. The relatively simple transactions in YCSB comprise a read-mostly workload on a large table with a single index. TPC-C has more complex transactions, many indexes, and many writes.

In all of our experiments, we measure the number of committed transactions, the number of index operations performed, the time needed to perform all transactions (*total time*), and the time spent accessing the index(es) (*index time*). Timing measurements were performed using x86-64 RDTSC instructions. The overall performance of a benchmark is measured in terms of *transaction throughput*, the total number of committed transactions divided by *total time*. We define *dbx time* as the time in an execution that is not spent accessing the index(es) (i.e., *total time*−*index time*).

4.1 YCSB

Following the approach in [39], we run a subset of the YCSB core with a single table containing ten million rows. Each thread performs a fixed number of transactions (100,000 in our runs), and the execution terminates when the first thread finishes performing its transactions. Each transaction accesses 16 different rows in the table, which are determined by index lookups on randomly generated keys. Each row is read with probability 0.9 and updated with probability 0.1. The keys are generated according to a Zipfian distribution following the approach in [19].

Segregating tree data When we use a BST implementation as the index in YCSB, we are effectively merging the memory address space of YCSB with the address space of the BST. In doing so, we may change the memory layout of objects in YCSB (for example, by interleaving nodes with table rows in YCSB), which can have a significant impact on performance. We can isolate and study these memory layout changes, and selectively eliminate them, by using *segregation* to effectively separate parts of the address spaces for the BST and YCSB.

In a real application, it can be difficult to segregate simply by changing object sizes, so we implement segregation by using *several separate instances* of the memory allocator: one for YCSB, and one for each type of objects we would like to segregate from other object types. In our case, this means one for BST nodes, one for BST descriptors, and one for other implementation specific tree data. Consequently, when we segregate tree data, nodes are allocated consecutively in each page, descriptors are *not* interleaved with nodes (avoiding the performance problem with *int-lf* in § 3.2.3), and tree data is *not* interleaved with YCSB data.

4.2 Comparison with the microbenchmark

We first address the question: to what degree do the results of YCSB match our microbenchmark results? We compare with microbenchmark results for trees containing 10 million keys, since this is approximately the size of the index in YCSB. The left side of Figure 5 contains the results of running the microbenchmark for all of the BSTs we studied, after fixing all of the performance issues described. To make the results easier to understand, we sort the BSTs by performance and group them into the following equivalence classes: (*occ-avl*, *occ-avl-2*), (*log-int*, *edge-int-lf*, *citrus*, *int-lf*, *edge-ext-lf-2*, *edge-ext-lf*), (*ticket*), (*ext-lf*). Within each of these equivalence classes, the performance differences are not significant.

The results of our YCSB experiments appear in Figure 5. The BSTs are listed in the same order as they appear in the microbenchmarks. Without segregation (middle

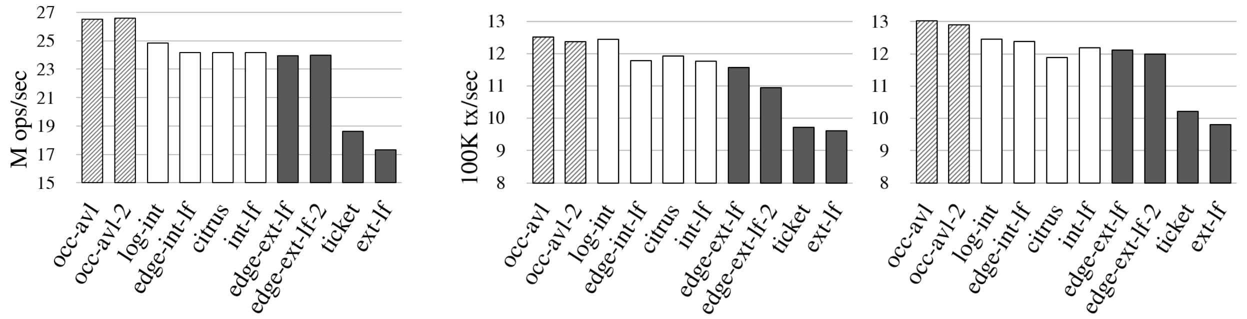


Figure 5: Microbenchmark compared to YCSB results: (left) Microbenchmark for 10M item BSTs (middle) YCSB *without* segregation, (right) YCSB *with* segregation.

graph) there are several differences between the YCSB results and the microbenchmark results. First, *log-int* performs about as well as *occ-avl* and *occ-avl-2*, which were significantly faster than *log-int* in the microbenchmarks. Here, it appears that *log-int* belongs in the same equivalence class as *occ-avl* and *occ-avl-2*. Second, *edge-ext-lf-2* is significantly slower than *edge-ext-lf*, whereas they have the same performance in the microbenchmark. Third, *ticket* and *ext-lf* have the same performance, whereas *ticket* is significantly faster in the microbenchmark. As the graph on the right shows, segregating the tree data bring the results closer to the original behaviour observed in the microbenchmark.

4.3 Memory layout issues

In our analysis of YCSB, we found several memory layout issues that were similar to the issues we found in our microbenchmarks. We describe a few key examples.

4.3.1 Underutilized caches due to allocation pattern

When we add all of our BST implementations to YCSB, several of them exhibit very poor cache set utilization. We find that their nodes map to only 1/3rd of the L3 cache sets, rendering 2/3rds of the L3 cache unusable for the storing nodes. These implementations include *occ-avl* and *occ-avl-2*, which have 64-byte nodes. Only implementations with 64-byte nodes were affected.

Since we did not observe this behaviour in the microbenchmarks, we hypothesize it is the result of adding these trees to YCSB (more specifically, merging each tree's memory space with the memory space of YCSB). We analyze the allocations performed by YCSB, and find that it allocates a large number (millions) of objects in size classes: 8, 32, 48, 64, 128, 192 and 384. In the 64-byte size class, it allocates only *row* and *row wrapper* objects. In particular, it always allocates a row, followed by a row wrapper, and then inserts the row into the index (BST). In the BSTs that exhibit this memory layout problem, index insertion allocates one 64-byte node. Thus, the allocation pattern in memory is RWNRWNRWN... where R is a row, W is a row wrapper, and N is a node. Consequently, rows have addresses satisfying $addr = 0 \ (mod\ 192)$, row wrappers have addresses satisfying $addr = 64 \ (mod\ 192)$ and nodes have addresses satisfying $addr = 128 \ (mod\ 192)$. That is, each object type has a 192-byte stride.

This pattern turns out to have a pathological interaction with the processor's internal hash function that maps physical addresses to L3 cache sets, resulting in an execution where rows, row wrappers and nodes each map to only 1/3rd of the L3 cache sets. (This is similar to how we saw a memory layout anomaly with a 128-byte stride in § 3.2.3.) In contrast, if a particular object type appears with a 256-byte stride, the L3 hash function will map objects approximately uniformly over all cache sets.

We break up this deleterious allocation pattern by segregating the tree data. This segregation results in a significant speedup for these data structures, since it allows nodes to occupy the entire cache. For example, in *occ-avl*, it reduces *index time* from 121 to 108 seconds (a 13 second difference), and *total time* from 188 to 178 seconds (a 10 second difference). Note, however, that it increases *dbx time* by 3 seconds. Further timing measurements demonstrate that the increase in *dbx time* is due to added contention on *row locks*. In fact, we can show that *whenever* segregation increased *dbx time* in YCSB, the increase is due to added contention on row locks.

Perhaps surprisingly, the deleterious allocation pattern we saw above did *not* affect *ticket*, which has 64-byte nodes, or a variant of *ext-lf* with 64-byte nodes. (These were the only other implementations with 64-byte nodes.) The explanation turns out to be fairly simple. Although their nodes are 64 bytes, these trees are external, so they allocate *two* nodes per insertion operation, producing the allocation pattern RWNNRWNNRWNN. The second node allocation breaks up the (pathological) 192-byte strides that we saw above.

4.3.2 Accidentally fixing a memory layout problem

In the previous section, we saw how merging two address spaces can cause a memory layout issue. In this section, we see how merging two address spaces can fix a preexisting memory layout issue.

When adding BSTs with 48-byte nodes to YCSB, and experimenting to see how much segregation helps, we find that segregating tree data for these BSTs caused significant *increases* in *dbx time*. For example, segregation increases *dbx time* for *edge-int-lf* by 9 seconds, from 58 to 67. Analyzing executions of YCSB, we find that approximately ten million *row locks* (implemented with `pthread` mutexes) and 4,000 other miscellaneous objects are allocated in the 48-byte size class (in addition to any 48-byte nodes).

If there are no 48-byte node allocations, then these 48-byte row locks experience false sharing. Since the locks are smaller than a cache line, and they are allocated consecutively, a single cache line contains parts of two different locks. Thus, write contention on one lock additionally creates write contention on another lock. This is exacerbated by the adjacent line prefetcher, which effectively causes accesses to a lock to contend with the (three to four) locks stored in *two* cache lines. By merging the address space of a BST with 48-byte nodes with the address space of YCSB, we accidentally mitigated this false sharing by interleaving row locks with nodes. Of course, this unfairly favours the BSTs with 48-byte nodes over the other BSTs. Thus, we fix this problem in a more principled way by padding the row locks to eliminate false sharing. (The same effect was seen, and fixed, in TPC-C.)

4.3.3 Unnecessary page scattering

So far, we have seen that segregation can improve performance by breaking up deleterious memory layouts, and generally improving cache behaviour. Our results have thus far suggested that we can reasonably expect to see some change in performance due to segregation whenever nodes are allocated from the same size class as some other objects. However, it turns out that segregation can improve performance, even when nodes are the only allocations performed from a given size class.

Once the row locks in YCSB are padded, YCSB only performs about 4,000 miscellaneous allocations from the 48-byte size class. Thus, in BSTs with 48-byte nodes, nodes are the only significant source of allocations in their size class. We were quite surprised to find that segregation significantly improved performance for these trees.

One interesting difference caused by segregation is a substantial reduction in the TLB miss rate for algorithms with 48-byte nodes. This improvement comes from an interaction between *huge pages* and the allocator. When huge pages are enabled in Linux, pages occupy 2MB instead of 4096 bytes. This generally improves TLB miss rates, since a program's working set can be represented using fewer pages.

However, we found that the allocator `jemalloc` divides each huge page into 512 chunks of 4096 bytes each, and distributes these into different size classes. More specifically, during its initialization, `jemalloc` allocates a bank of chunks for each thread. Each thread distributes these chunks *on-demand* to its individual size classes. Whenever a thread runs out of space in the current chunk for one of its size classes, it fetches one (or more) chunks from its bank, and assigns them to this size class. In our experiments, we observed that threads fetch one chunk at a time for the 32-, 48- and 64-byte size classes.

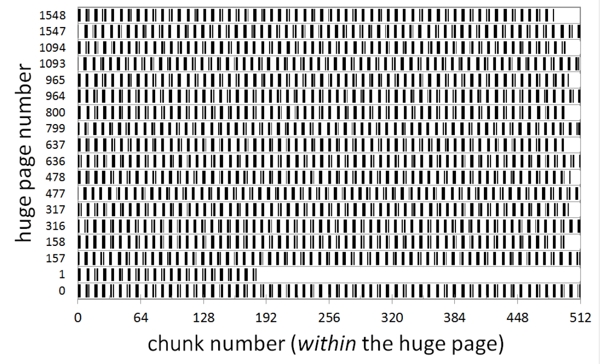


Figure 6: Layout of pages *without* segregation: all *chunks* used to store nodes by *occ-avl-2* in YCSB (*haswell*).

In YCSB, this has the following effect. Before performing insertion on a BST, a transaction allocates a 64-byte row, followed by 128 bytes of data, a 64-byte row wrapper, a 192-byte (padded) `pthread` mutex, and a 32-byte value. Thus, for each node allocated by an insertion operation, several objects are allocated in several different size classes. Consequently, each thread regularly takes chunks from its bank and assigns them to these size classes, almost in round-robin fashion, but with more chunks going to the size classes that exhaust them more quickly.

As a result, in the small size classes used for nodes, the chunks often do *not* have consecutive addresses. For example, in a variant of *edge-int-lf* with 48-byte nodes, we found that threads would allocate full 4096-byte chunks of nodes, but would only store nodes in approximately one out of every 10 chunks that it allocated. As another example, in *occ-avl-2*, which has 64-byte nodes, threads would use up to three consecutive chunks to store nodes, and then the next chunk used to store nodes would typically appear five or six chunks later in the address space. Figure 6 visualizes the actual layout of chunks used to store nodes in an execution of YCSB with *occ-avl-2*.

We now consider what happens when the tree data for *occ-avl-2* is segregated. Figure 7 shows the resulting layout of chunks used to store nodes. The difference is striking. Since the nodes are allocated by a separate instance of `jemalloc`, each thread uses its entire bank of chunks to store nodes. Consequently, the chunks allocated for nodes almost always have consecutive addresses. This

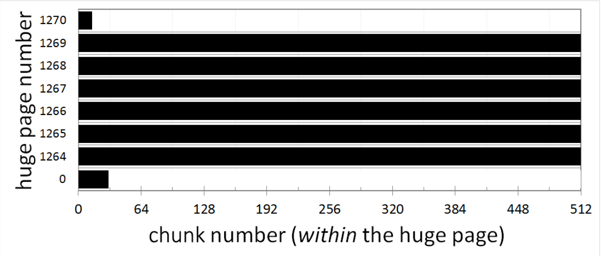


Figure 7: Layout of pages *with* segregation: all *chunks* used to store nodes by *occ-avl-2* in YCSB (*haswell*).

significantly reduces the number of pages needed to store the tree, and results in far fewer TLB misses. In YCSB with *occ-avl-2*, the average number of TLB misses per YCSB transaction decreases from 412 to 161 (a 61% reduction). Segregation reduces TLB misses in all of the BSTs we studied.

4.4 TPC-C

TPC-C simulates a large scale online transaction processing application for the order-entry environment of a wholesale supplier. According to the Transaction Processing Performance Council, it represents the business activity of "any industry that must manage, sell, or distribute a product or service." At a high level, TPC-C assumes that business operations are organized around a fixed number of warehouses, which each service a number of districts. For each warehouse and district, the database stores information about customers, orders, payments, items for sale, and warehouse stock. TPC-C features complex transactions over nine tables with widely varying row types and population sizes, and with varying degrees of non-uniformity in the data. These tables are indexed by up to three different indexes on different key fields.

Our implementation of TPC-C executes a representative subset of the TPC-C transactions. In particular, we include the *new-order* and *payment* transactions, which comprise 88% of all transactions executed in the full TPC-C benchmark. This same approach was taken in [39].

Note that payment transactions update data in the *warehouse* table, and thus contend with all transactions operating on the same warehouse. Consequently, concurrency in TPC-C is limited by the number of warehouses. Thus, it is common to run with at least as many warehouses as there are concurrent threads in the experimental system. We run with 48 warehouses.

Segregating tree data As in YCSB, we segregate the tree data by using several allocator instances: one for TPC-C, one for BST nodes, one for descriptors, and one for other implementation specific tree data. All indexes share the same allocators. So, for example, all indexes use the same allocator for nodes. Thus, nodes for all indexes are interleaved with one another, but not with TPC-C data.

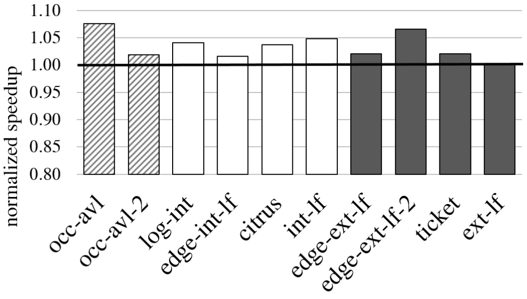


Figure 8: TPC-C: baseline vs. improved implementations.

4.5 Impact of improved BSTs on TPC-C

We now present an experiment that demonstrates the impact of our improvements to the BSTs on the performance of TPC-C. The results appear in Figure 8. We obtain each data point by dividing the throughput of TPC-C when the final BST implementation is used for indexes (with segregation) by the throughput when the baseline BST implementation (without segregation) is used for indexes.

By improving the BST implementations, we obtain an overall improvement of up to 7.6%. Initially, this improvement might seem somewhat small, but TPC-C is a large, complex workload that takes over 200 seconds to run, and allocates over 30 GiB of memory. Accesses to the indexes comprise a relatively small part of the work, and Amdahl's law limits the improvement we can see, so a 7.6% overall improvement is actually fairly substantial.

Source of the improvement Let us drill down into the details of where this improvement comes from. As an example, we consider *occ-avl* (which obtains the full 7.6% improvement). With the baseline implementation of *occ-avl*, the *total time* to run TPC-C is 249 seconds. This breaks down into 108 seconds of *index time* and 141 seconds of *dbx time*. If we follow the recommendations in § 3, then *total time* decreases by 9 seconds to 240. This breaks down into 105 seconds of *index time* and 135 seconds of *dbx time*. If we additionally segregate tree data, then *total time* further decreases by 8 seconds to 232. This breaks down into 95 seconds of *index time* and 137 seconds of *dbx time*.

Interestingly, segregation causes a slight increase in *dbx time*. It turns out that, when the indexes in DBx speed up significantly, a new bottleneck appears. This manifests as increased contention on row locks. However, this is not the only component of the increase in *dbx time*. DBx and TPC-C are quite complex, and there is an additional component that we are unable to identify. We leave it as future work to perform additional profiling of DBx.

4.6 Impact of segregation on TPC-C

We now study the effect of segregation on the other BSTs. Figure 9 shows the breakdown of TPC-C *total time* into *index time* and *dbx time* both with and without segregation

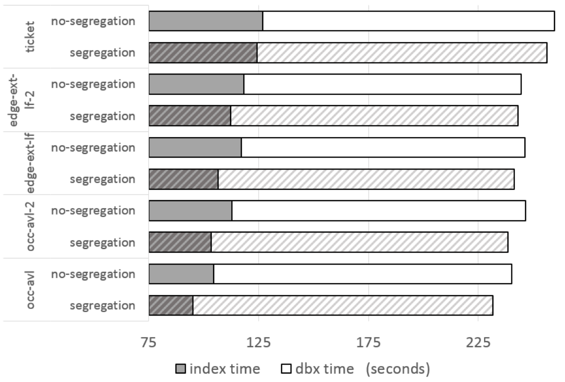


Figure 9: Impact of segregation on TPC-C.

of tree data. Note that the x-axis starts at 75 seconds. The BSTs that are not shown in the graph do not experience significant changes in either *index time* or *dbx time*.

As we saw above, segregation improves the *index time* of *occ-avl* by 10 seconds, and *hurts* its *dbx time* by 2 seconds. That is, it helps *index time* much more than it hurts *dbx time*. In contrast, consider *edge-ext-lf*, for which segregation improves *index time* by 10 seconds, but *hurts dbx time* by 6 seconds, negating most of the benefit. In this case, approximately 2 seconds of the change in *dbx time* is due to increased contention on row locks.

Although the benefit of segregation is somewhat limited in Figure 9, it is important to remember that we are starting from optimized implementations that follow the recommendations in § 3. Different implementations will interact with TPC-C's memory layout in different ways, and may see more significant benefits. For example, we ran TPC-C with a variant of *int-lf* that has 64-byte nodes and 112-byte descriptors, instead of 48-byte nodes and 64-byte descriptors. For this BST, segregation does increase *dbx time* by 5 seconds from 135 to 140, but it greatly improves *index time* by 16 seconds from 113 to 97.

5 Related work

Memory layout issues Some of the phenomena we find are reported in other contexts [1, 28, 37], but these works do not consider the combination of all factors and their effect on BST performance. Earlier research proposed compiler and library techniques for improving cache utilization by careful placement of objects in memory [8, 9], but these techniques are not deployed and so it is not clear whether they would address the anomalies we consider.

Segregated allocations Region-based memory management [20, 34] allocates each object type from a dedicated memory pool. However, its motivation is to speed up memory allocation and freeing, not to improve cache and TLB utilization. Lattner and Adve [25] propose a compiler algorithm for segregating distinct instances of data structures into separate pools. Their approach does not segregate allocations within a data structure, which may be required to avoid underutilizing cache sets.

Understanding performance Several studies compare the performance of concurrent data structures [12, 18], but do not analyze the root causes of performance differences. Our work is complementary to research on the difficulties of understanding experimental evaluation results [6, 13, 31], which does not consider concurrent data structures.

6 Discussion

We believe that the lessons learned in this work can be applied to other concurrent data structures, as they stem from general performance principles. Here, we attempt to distill these lessons into concrete recommendations.

Data structure designers and implementers: Study the memory layout of the data structure. If cache lines adjacent to nodes often contain other objects, then the cache may be underutilized by nodes. Pad objects to separate them into different allocator size classes. Padding should also be used to avoid false sharing, particularly between frequently-accessed nodes and other program data. Such padding should take prefetching (e.g., the adjacent line prefetcher) into account. However, indiscriminately padding *all* nodes may reduce performance, since this reduces the number of nodes that fit in the LLC. Finally, watch for and avoid the implementation problems in § 3.1.

Programmers using a data structure: Importing a data structure into a program merges two memory spaces, and may create *or eliminate* false sharing or cache underutilization problems. Thus, one should either (a) inspect the combined memory layout of the data structure and the program, and fix such problems, or (b) segregate the data structure's memory by using a separate allocator.

Memory allocator designers and implementers: The above recommendations would be substantially easier to put into practice with additional support from memory allocators: First, providing an interface for *allocation segregation*. Second, providing interfaces or tools for *memory layout inspection*, to allow determining (1) the mapping of objects to size classes; (2) which object types are frequently located *close* to one another in memory (where *close* could mean in the same cache line, or in adjacent cache lines, or in the same page); and (3) the distribution of objects into cache sets in the LLC (for each object type). Such queries could also lead to high quality automated tools for identifying memory layout problems.

Acknowledgments

This work was funded in part by the ISF (grants 2005/17 & 1749/14). Maya Arbel-Raviv was supported in part by the Technion Hasso Platner Institute Research School. Trevor Brown was funded by PhD and post-doctoral fellowships from NSERC, and grant no. RGPIN-2015-05080. Adam Morrison was supported by Len Blavatnik and the Blavatnik Family Foundation.

References

[1] Afek, Y., Dice, D., and Morrison, A. Cache Index-Aware Memory Allocation. In *ISMM* (2011).

[2] Arbel, M., and Attiya, H. Concurrent Updates with RCU: Search Tree As an Example. In *PODC* (2014).

[3] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS* (2012).

[4] Avni, H., and Brown, T. Persistent Hybrid Transactional Memory for Databases. *VLDB 10*, 4 (Nov. 2016).

[5] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *ASPLOS* (2000).

[6] Blackburn, S. M., McKinley, K. S., Garner, R., Hoffmann, C., Khan, A. M., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanovik, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *CACM 51*, 8 (Aug. 2008).

[7] Bronson, N. G., Casper, J., Chafi, H., and Olukotun, K. A Practical Concurrent Binary Search Tree. In *PPoPP* (2010).

[8] Calder, B., Krintz, C., John, S., and Austin, T. Cache-conscious Data Placement. In *ASPLOS* (1998).

[9] Chilimbi, T. M., Hill, M. D., and Larus, J. R. Making Pointer-Based Data Structures Cache Conscious. *Computer 33*, 12 (Dec. 2000).

[10] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. Scalable address spaces using RCU balanced trees. In *ASPLOS* (2012).

[11] Crain, T., Gramoli, V., and Raynal, M. A Contention-friendly Binary Search Tree. In *Euro-Par* (2013).

[12] David, T., Guerraoui, R., and Trigonakis, V. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS* (2015).

[13] de Oliveira, A. B., Fischmeister, S., Diwan, A., Hauswirth, M., and Sweeney, P. F. Why You Should Care About Quantile Regression. In *ASPLOS* (2013).

[14] Drachsler, D., Vechev, M., and Yahav, E. Practical Concurrent Binary Search Trees via Logical Ordering. In *PPoPP* (2014).

[15] Ellen, F., Fatourou, P., Ruppert, E., and van Breugel, F. Non-blocking Binary Search Trees. In *PODC* (2010).

[16] Evans, J. Scalable memory allocation using jemalloc. http://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919, 2011.

[17] Ghemawat, S., and Menage, P. TCMalloc: Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[18] Gramoli, V. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP* (2015).

[19] Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. J. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1994), SIGMOD '94, ACM, pp. 243–252.

[20] Hanson, D. R. Fast allocation and deallocation of memory based on object lifetimes. *SPE 20*, 1 (1990).

[21] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[22] Howard, P. W., and Walpole, J. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience* (2013).

[23] Howley, S. V., and Jones, J. A Non-blocking Internal Binary Search Tree. In *SPAA* (2012).

[24] Johnstone, M. S., and Wilson, P. R. The memory fragmentation problem: solved? In *ISMM* (1998).

[25] Lattner, C., and Adve, V. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI* (2005).

[26] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. SILT: A Memory-efficient, High-performance Key-value Store. In *SOSP* (2011).

[27] Lim, H., Kaminsky, M., and Andersen, D. G. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD* (2017).

[28] Lvin, V. B., Novark, G., Berger, E. D., and Zorn, B. G. Archipelago: trading address space for reliability and security. In *ASPLOS* (2008).

[29] Mao, Y., Kohler, E., and Morris, R. T. Cache Craftiness for Fast Multicore Key-value Storage. In *EuroSys* (2012).

[30] Maurice, C., Scouarnec, N. L., Neumann, C., Heen, O., and Francillon, A. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *RAID* (2015).

[31] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. Producing Wrong Data Without Doing Anything Obviously Wrong! In *ASPLOS* (2009).

[32] Natarajan, A., and Mittal, N. Fast Concurrent Lock-free Binary Search Trees. In *PPoPP* (2014).

[33] Ramachandran, A., and Mittal, N. A Fast Lock-Free Internal Binary Search Tree. In *ICDCN* (2015).

[34] Ross, D. T. The AED Free Storage Package. *ACM 10*, 8 (Aug. 1967).

[35] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. Speedy Transactions in Multicore In-memory Databases. In *SOSP* (2013).

[36] Viswanathan, V. Disclosure of H/W prefetcher control on some Intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, 2014.

[37] Wu, C.-J., and Martonosi, M. Characterization and Dynamic Mitigation of Intra-application Cache Interference. In *ISPASS* (2011).

[38] Yarom, Y., Ge, Q., Liu, F., Lee, R. B., and Heiser, G. Mapping the Intel Last-Level Cache. Cryptology ePrint Archive, Report 2015/905, 2015. http://eprint.iacr.org/2015/905.

[39] Yu, X., Bezerra, G., Pavlo, A., Devadas, S., and Stonebraker, M. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *VLDB 8*, 3 (Nov. 2014).

[40] Yu, X., Pavlo, A., Sanchez, D., and Devadas, S. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD* (2016).

TerseCades: Efficient Data Compression in Stream Processing

Gennady Pekhimenko
University of Toronto

Chuanxiong Guo
Bytedance Inc.

Myeongjae Jeon
Microsoft Research

Peng Huang
Johns Hopkins University

Lidong Zhou
Microsoft Research

Abstract

This work is the first systematic investigation of stream processing with data compression: we have not only identified a set of factors that influence the benefits and overheads of compression, but have also demonstrated that compression can be effective for stream processing, both in the ability to process in larger windows and in throughput. This is done through a series of (i) optimizations on a stream engine itself to remove major sources of inefficiency, which leads to an order-of-magnitude improvement in throughput (ii) optimizations to reduce the cost of (de)compression, including hardware acceleration, and (iii) a new technique that allows direct execution on compressed data, that leads to a further 50% improvement in throughout. Our evaluation is performed on several real-world scenarios in cloud analytics and troubleshooting, with both microbenchmarks and production stream processing systems.

1 Introduction

Stream processing [7, 18, 22, 41, 1, 10, 11, 66, 48, 56, 57, 65, 21, 17] is gaining popularity for continuous near real-time monitoring and analytics. It typically involves continuous processing of huge streams of machine-generated, timestamped measurement data. Examples include latency measurements [35], performance counters, and sensor readings in a wide variety of scenarios such as cloud systems and Internet of Things (IoT) [2, 3]. In order to meet near real-time requirements, stream processing engines typically require that streaming data (coming in huge volumes) reside in the main memory to be processed, thereby putting enormous pressure on both the capacity and bandwidth of the servers' main memory systems. Having high memory bandwidth while preserving capacity is known to be difficult and costly in modern DRAM [44, 19, 63]. It is therefore important to explore ways, such as data compression, to relieve this memory pressure.

This paper presents the first systematic investigation of stream processing with data compression. The low-latency, mostly in-memory processing characteristics make data compression for stream processing distinctly different from traditional data compression. For example, in database or (archival) file systems, a sophisticated compression scheme with high compression ratio [68, 27, 37, 47] is often desirable because its overhead can be overshadowed by high disk latency. We start by observing opportunities for significant (orders of magnitude) volume reduction in production cloud measurement data streams and real-world IoT data streams, processed in real stream queries for cloud analytics and troubleshooting purposes, as well as for IoT scenarios. The high redundancy in the streaming data sets is primarily due to the synthetic and numerical nature of these data sets, including, but not limited to, timestamps, performance counters, sensor, and geolocation data. This key observation creates an opportunity to explore efficient encoding mechanisms to explore streaming data redundancy, including lossless and lossy compression (that is *harmless* with respect to specific queries output). For example, timestamps in the data streams are highly compressible even through simple lossless encoding mechanisms, such as variable-length coding [54] and base-delta encoding [53, 60]. By knowing the semantics of generic window-based streaming operators, we can further improve the benefits of compression by reducing the over-provisioning of the timestamps accuracy *without* affecting the produced results. The potential we have identified (for several representative streaming datasets) is likely to apply to other machine-generated time series as well.

Volume reduction, however, does not necessarily lead to proportional improvement in end-to-end throughput, even on a state-of-the-art stream engine such as Trill [21]. Our evaluation shows that an 8× reduction in data volume translates to less than 15% improvement in throughput on Trill, even without considering any encoding cost. This is because memory bandwidth is not yet the bottleneck thanks to significant overhead elsewhere in the system. We therefore build TerseCades[1], a lean stream processing library that optimizes away the other major bottlenecks that we have identified in the existing stream engines, using techniques such as array reshaping [67], static memory allocation and pooling [42, 26], and hashing. TerseCades provides a vastly improved and competitive baseline (by an order of magnitude in throughput over Trill), while making a strong case for compression in streaming context.

Driven by real streaming queries on production data, we have identified factors that influence the benefits and overheads due to data compression, and proposed a series of optimizations to make compression effective for stream processing. This includes the use of SIMD instructions for data compression/decompression, hardware acceleration (using GPUs and FPGAs), as well as supporting execution directly on compressed data when it is feasible. To demonstrate the end-to-end benefits of our design, we have implemented compression support with the optimizations on TerseCades. Our evaluation shows that, altogether, these optimizations can improve the throughput by another 50% in TerseCades, on top of the order of magnitude improvement over Trill, while significantly improving processing capacity diverse temporal windows due to reduced memory footprint.

In summary, our contributions are as follows. (1) We identify major bottlenecks in a state-of-art stream engine and develop TerseCades that provides an order of magnitude higher throughput. (2) We characterize representative data streams and present compression algorithms for effective in-memory stream processing. (3) We implement these compression algorithms along with a set of optimizations (e.g., direct execution on compressed data) on TerseCades, improving throughput by another 50%.

2 Is Compression Useful for Streaming?

A natural starting point to assess the usefulness of compression for streaming is to check (i) whether data streams are compressible and (ii) whether data volume reduction from compression improves the throughput of stream processing. To do so, we perform a set of analysis using the Pingmesh data streams of network reachability measurements [35] from production data centers, with respect to motivating real data set for data-center network diagnosis. We then use Trill [21], a state of the art high-performance streaming library, and the STREAM [13] benchmark suite to evaluate the effect of data volume reduction on throughput.

2.1 Streaming Data Compressibility

For compressibility, we examine the Pingmesh data records. Major fields are listed in Table 1, and here we focus on two important fields: (i) 8-byte integer `timestamp` to represent the time when the request was issued, and (ii) 4-byte integer `rtt` values to represent request round-trip-time (in microseconds).

Stream processing operates on batches of data records that form *windows*. Our analysis on those batches reveals the potential of significant volume redundancy that can be easily exploited. For example, the `timestamp` values are often within a small range: more than 99% of the values in a 128-value batch differ in only 1 lower-order byte. This potentially allows efficient compression with simple lossless compression schemes such as Base-Delta encoding [53, 60] and variable-length coding [54] to achieve a compression ratio around 8× or more. Similarly, the `rtt` values for successful requests are usually relatively small: 97% values need only two bytes. This data can be compressed by at least a factor of 2.

While lossless compression can be effective in reducing data redundancy, we observe that in many real scenarios it is profitable to explore *lossy* compression without affecting the correctness of the query results. For example, in queries where timestamps are used in a windowing operator only for assigning a record to a time window, we can replace multiple timestamps belonging to the same window with just one value that maps them to a particular window. We provide more details on lossy compression in Section 3.3.

2.2 Compressibility $\not\Rightarrow$ Performance Gain

We further study the effect of data volume reduction on stream processing using Trill, driven by a simple *Where* query that runs a filter operator on a single in-memory data field. We use two versions of the data field (8 and 1 bytes) to simulate an ideal no-overhead compression with a compression ratio of 8. This query performs minimum computation, does not incur any compression/decompression overhead, allowing Trill to focus on the actual query computation. Figure 1 shows the results.[2]

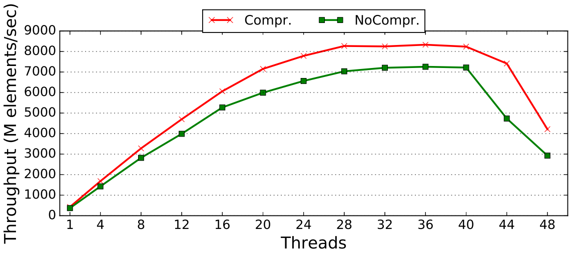


Figure 1: Throughput with data compression in Trill.

[1] TerseCades = Terse (for compression) + Cascades (for streaming). Appropriately several characters in Cascades get 'compressed'.

[2] Section 4 describes the methodology and system configurations.

As expected, the 1-byte compressed (*Comp*) version consistently outperforms the 8-byte non-compressed (*NoComp*) version. However, the amount of the improvement is relatively small (only 13-15%), compared to a factor of 8 reduction in memory traffic. This indicates that query processing in Trill is not memory-bound even when the query executes simple computation (e.g., a filter operator).

To understand the source of such inefficiency, we have run and profiled a diverse set of queries (including filter query and groupby query) using Trill. Our profiling shows that for the filter query (and similarly for other stateless queries), most of the execution time is spent in functions that generalize semantics in streaming data construction. In particular, for each incoming event, the filter query (1) performs just-in-time copy of payloads to create a streameable event (memory allocation) and (2) enables flexible column-oriented data batches (memory copying and reallocation). These operations account for more than two-thirds of the total query processing time, with limited time spent on the query operator itself. The second major overhead is inefficient bit-wise manipulation; 46% of the time is spent on identifying what bit should be set when the previous bottleneck is fully removed. For the groupby query, more than 90% of the time is spent on manipulating the hash table (e.g., key lookups), which holds the status of the identified groups. While adding concurrency mitigates such costs, they remain the largest.

In summary, we conclude that the state-of-the-art streaming engines such as Trill are not properly optimized to fully utilize the available memory bandwidth, limiting the benefit of reduced memory consumption through data compression. In our work, we will address this issue by integrating several simple optimizations that make streaming engines much more efficient and consequently memory sensitive (Section 3.1).

2.3 Compressibility ⇒ Performance Gain

To understand whether the limitations we observe with Trill are fundamental, we look at the performance of the STREAM [13] benchmark suite, which performs simple streaming operators such as copy, add, and triad on large arrays of data[3], without the overhead we observe in Trill.

Figure 2 shows the throughput of the *Add* benchmark for three different cases: (i) *Long* – 64-bit unsigned integer, (ii) *Char* – 8-bit char type (mimic 8× compression with *no* compression/decompression overhead, (iii) *CharCompr.* – compressing 64-bit values to 8-bit using Base-Delta encoding [53].

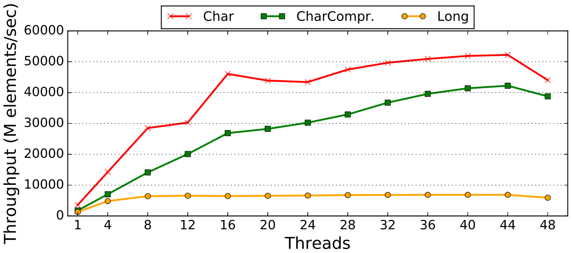


Figure 2: Add results.

We draw two major conclusions from this figure. First, when the amount of data transferred is reduced 8× (going from Long to Char), the resulting throughput also increases proportionally, from the maximum of around 7 Billion elements/sec for *Long* all the way to the maximum of 52 Billion elements/sec with *Char*). The significant throughput improvement indicates that, due to the absence of other artificial bottlenecks (e.g., memory allocation/deallocation, inefficient bit-wise manipulation, and hashmap insertions/searches), the throughput of this simple streaming engine is limited only by the main memory bandwidth, and compression reduces the bandwidth pressure proportionally to the compression ratio. Second, using a realistic simple compression mechanism, e.g., *CharCompr.* with Base-Delta encoding, still provides a lot of benefits over uncompressed baseline (the maximum increase in throughput observed is 6.1×), making simple data compression algorithms an attractive approach to improve the performance of stream processing. At the same time, it is clear that even simple compression algorithms incur noticeable overhead that reduces the benefits of compression, hence the choice of compression algorithm is important.

3 Efficient Compression with TerseCades

In this section we first describe the key optimizations (which we refer to as first-order) that are needed for a general streaming engine to be efficient. We then describe the design of a single-node streaming system that supports generic data compression. Finally, we show the reasons behind the choice of compression algorithms we deploy in TerseCades, hardware-based strategies to minimize the (de)compression overhead (using SIMD, GPU and FPGA), as well as less intuitive (but very powerful) optimizations such as direct execution on compressed data.

3.1 Prerequisites for Efficient Streaming

Our initial experiments with Trill engine (§2) show that in order to make streaming engines more efficient, several major bottlenecks should be avoided. First, dynamic memory allocation/deallocation is costly in most operating systems, and permanent memory allocation for every

[3] For our purposes, we evaluate STREAM not only on float/double data, but also for different integer types.

window (or even batch within a window) in streaming engine significantly reduces the overall throughput. This happens because the standard implementation of streaming with any window operator would require dynamic memory allocation (to store a window of data). One possible strategy to address this problem is to identify how much memory is usually needed per window (this amount tends to be stable over time as windows are normally the same size), and then use *fixed memory allocation* strategy – most of the memory allocation happens once and then reused from the internal memory pool. In TerseCades we use profiling to identify how much memory is usually needed for a particular size window, allocate all this memory at the beginning, and then only allocate more memory if needed during the execution.

Second, implementation of certain streaming operators, e.g., *GroupApply*, requires frequent operation on hashmap data structures. Similarly, many common integral data types such as strings, might require a lot of memory if stored fully (e.g., 64 bytes for the server IDs), but can be efficiently hashed to reduce space requirements. Unfortunately, the standard C++ STL library does not provide this efficiency. To address this problem, we implement our own hashmap data structure with corresponding APIs taking into the account specifics of our streaming data.

Third, efficient implementation of filtering operators (e.g., *Where*) requires efficient bit-vector manipulation. For example, when running a simple *Where* query with a single comparison condition (e.g., `Where (error Code == 0)`) with Trill streaming engine, we observe that about 46% of the total execution time is now related to simple bit-wise manipulation (1 line of the source code using standard C# data structures). Unfortunately, this huge overhead limits the benefits of any further performance optimizations. In our design, we implemented our own simple bit-wise representation (and the corresponding APIs) for filtering operators using C++ that significantly reduces the overhead of filtering. Altogether, these optimizations allows us to improve the performance our system more than 3× as we will show in Section 5.

3.2 System Overview

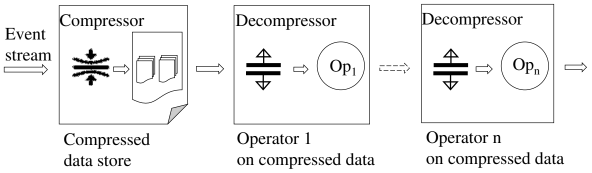


Figure 3: The streaming processing pipeline with compression and decompression.

Figure 3 shows our proposed TerseCades streaming processing pipeline in a single node. We will defer the discussion on how TerseCades is applied in the distributed system setting for monitoring and troubleshooting for a large cloud provider in Section 5.4, and in this section we focus on making this system efficient. We also note that single-node TerseCades system is generic, and both its design and optimizations behind it can be applied in other distributed streaming systems.

The major difference from traditional streaming processing, in Figure 3, is that external streaming events are first compressed before they are stored (typically in a column-oriented format that is usually more preferable for applications with high spatial locality). Note that the streaming operators also need to carry out decompression on all the compressed data before they access it (except for the cases where we use direct execution on compressed data described in Section 3.3).

The operators are chained together to form a pipeline. Different operators may work on different columns of the data, hence they may need to perform different decompression operations. Furthermore, some operators may need to produce new data sets from their input, and the newly generated data sets need to be compressed as well. This flow highlights the fact that compression/decompression operations are now on the critical path of the streaming engine execution, and have to be efficient to provide any benefits from tighter data representation .

3.3 Practical Compression for Streaming

One of the key contributions of this work is the efficient utilization of the existing memory resources (both bandwidth and capacity) by using simple yet efficient data compression algorithms. We observe that the dominant part of the data we use in stream processing is synthetic in nature, and hence it has a lot of redundancy (see Section 2 and 5) that can be exploited through data compression. In this section, we describe the key design choices and optimizations that allowed us to make data compression practical for modern streaming engines.

Lossless Compression. The key requirement of lossless compression is that the data after decompression should be exactly the same as before compression. The classical lossless compression algorithms include different flavors of Lempel-Ziv algorithm [68], and Huffman encoding [37, 27, 28, 47] and arithmetic coding [60, 36, 54, 4, 59]. These algorithms were proven to be efficient for disk/storage or virtual memory compression [62, 29, 9] and graphics workloads [60, 36, 54], but unfortunately most of these algorithms are too slow for compressing active data in memory. [4] As we will show in Section 5.1,

[4] Memory latencies are usually on the order of tens of nanoseconds [39]. Even when these algorithms were implemented as an ASIC design, e.g., IBM MXT design [8, 61], the overhead more than double the latency for main memory accesses.

software implementations of these algorithms are usually impractical.

To address this challenge, rather than using sophisticated dictionary-based algorithms, we decided to use simple arithmetic compression algorithm that was recently proposed in the area of computer architecture – Base-Delta encoding [53]. The primary benefits of this algorithm include its simplicity (e.g., only one addition is needed for decompression), and its competitive compression ratio for a wide range of data (e.g., rtts, timestamps, pixels, performance counters, geolocation data). Figure 4 shows how timestamp data can be compressed with Base-Delta encoding (8-byte base and 1-byte deltas).

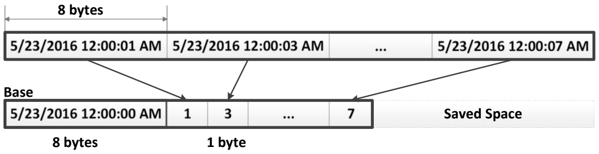


Figure 4: Base-Delta encoding applied to timestamps.

It turns out that this simple algorithm has several other benefits. First, it can be easily extended to a more aggressive lossy version that can still provide the output results that match lossless and uncompressed versions. Second, this algorithm is amenable to hardware acceleration using existing hardware accelerators such as GPUs and FPGAs, and using SIMD vector instructions available in commodity CPUs. Third, Base-Delta encoding preserves certain properties of the data (e.g., order) that can enable further optimizations such as direct execution on compressed data (Section 3.3).

Lossy Compression without Output Quality Loss. Lossy compression is a well-known approach to increase the benefits of compression at the cost of some precision loss. It is efficiently used in the areas where there is a good understanding of imprecision effect on the output quality, e.g., audio encoding, image compression. We observe that similar idea can be useful for some common data types in stream processing when the data usage and the effect of imprecision is also well understood.

For example, in troubleshooting scenario (§5.1), every record has a `timestamp` (8-byte integer) that is usually used only to check whether this timestamp belongs to a particular time window. As a result, storing the precise value of the timestamp is usually not needed and only some information to check whether the record belongs to a specific window is needed. Luckily, if the value already compressed with Base-Delta encoding, this information is usually already stored in the *base* value. Hence, we can avoid storing the *delta* values in most cases and get much higher compression ratio. For a batch of 128 `timestamp` values, the compression ratio can be as high as 128× for this data field, in contrast to about 8× compression ratio with lossless version. While this approach is not applicable in all cases, e.g., server IDs need to be precise, its impressive benefits while preserving the output quality made us consider using modified (lossy) version of Base-Delta encoding in our design.

Lossy Compression for Floating Point Data. The nature of floating point value representation makes it difficult to get high compression ratio from classical Base-Delta encoding. Moreover, getting high compression ratio with lossless compression algorithms on floating point data is generally more difficult [20, 14, 15]. Luckily, most of the scenarios using floating point values in streaming do not usually require perfect accuracy. For example, in several scenarios that we evaluated (§5), floating point values are used to represent performance counters, resource utilization percentage, geolocation coordinates, and sensor measurements (e.g., wind-speed or precipitation amount). In these cases, you usually do not need the precise values for all data fields to get the correct results, but certain level of precision is still needed.

We consider two major alternatives: (i) fixed point representation that essentially converts any floating point value into an integer value and (ii) using *lossy* floating point compression algorithms (e.g., ZFP [45]). The primary advantage of the first option is low overhead compression/decompression, because we can use Base-Delta encoding to compress the converted values. The primary benefit of the lossy floating point compression algorithms is that they usually provide higher accuracy than fixed-point representation. The lossy compression algorithm, called ZFP [45] that we use in our experiments, has the option to provide an accuracy bound for every value compressed. This option simplifies the usage of lossy compression since we only need to reason about data accuracy in simple terms (e.g., error bound per value is 10^{-6}). Moreover, this algorithm proved to have a very competitive throughput for both compression and decompression and allows to access the compressed data without decompressing it fully. Hence, in our design, we decided to use ZFP algorithm for floating point data.

Reducing the Compression/Decompression Cost. As we show in Section 2.3, even simple compression algorithms like Base-Delta encoding can add significant overhead. In this section, we will demonstrate how those overheads can be significantly reduced if we use the existing hardware to accelerate the major part of this overhead – data decompression.

Acceleration using SIMD instructions. Our original software implementation of Base-Delta encoding algorithm uses a simple add instruction to decompress a value based on its corresponding base and delta values. In streaming, usually many values are accessed at the same time, hence it is possible to reduce decompression overhead by using SIMD instructions, e.g., Intel AVX instructions (256-bit versions are available on most modern CPUs). By using SIMD instructions, we can reduce the overhead of decompression at least 4×, as four decompressions can be replaced with a single one. As we will show in Section 4.2, this optimization can significantly reduce the overhead of decompression that leads to the throughput close to the ideal compression case (with no compression/decompression overhead).

Hardware Acceleration: GPUs/FPGAs. Modern hardware accelerators such as Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) can be very efficient computational substrate to perform bulk compression/decompression operations [50, 31]. These accelerators are now available in commodity data centers for general-purpose use [55, 12, 16]. In our work, we also evaluate such a possibility by implementing Base-Delta encoding algorithm using CUDA 8.0 [49] on a GPU and using SystemVerilog on an FPGA. Our results (see Section 4.2) shows that by utilizing these accelerators, it is possible to perform the required data decompression (and potentially compression) without slowing down the main computation.

Direct Execution on Compressed Data. Many data compression algorithms require compressed data to be decompressed before it is used. However, performing each operation after data decompression can potentially lead to significant performance penalty. In streaming analytics, many operations are relatively simple and regular, allowing direct execution on the *compressed data itself.*[5] We find that we can run a set of stateless operators (e.g., Where) as well as aggregation operators (e.g., sum, min/-max, average, standard deviation, argmax/argmin, countif, distinct count, percentiles) on top of compressed data (assuming Base-Delta Encoding) more efficiently.

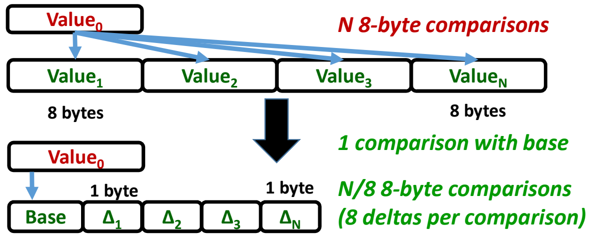


Figure 5: Direct execution on the data compressed with Base-Delta encoding.

Consider a simple *Where* query that performs a linear scan through an array on *N* values searching for a certain $value_0$ (see Figure 5). If this data is already compressed with Base-Delta encoding, then one simple strategy is to try to represent the searched value in the same base-delta format as batches of values in this array. If $value_0$ cannot be represented in this form (one comparison needed to test this), then this value is not in this batch, and there is no need to do any per-value comparisons. This avoids multiple (e.g., hundreds) comparisons per batch.

In cases where the search value can be represented similarly to the values in the batch, we still need to do value-by-value comparisons, but these values are now stored in a more narrow format (1-byte deltas in Figure 5). Instead of 8-byte comparisons, we can now group the deltas to do more efficient comparisons using SIMD instructions. This would reduce the number of comparisons by 8×. In summary, we can significantly improve the execution time for some common queries by utilizing the fact that data is stored in a compressed format to perform some operations more efficiently.

Generality of the Proposed Approach. There is a potential concern with the proposed approach due to limited generality and whether the benefits will be preserved when additional layers of indirection are added. Our current implementation supports a subset of queries from a LINQ-like proviver (e.g., *Where*, *Window*, *GroupBy*, *Aggregate* wih different operators inside of it, *Reduce*, *Process* and other operators are already supported), and is designed in a way, where it is possible to add support for new operators without significantly affecting the performance of existing ones. We currently leave it to the programmer to decide on whether they want their data being compressed using different proposed compression algo- rithms (similarly to how data compression is supported in Oracle databases). Our design favors column-oriented data allocation as it allows to get higher benefit from data locality in both DRAM and caches. It might not be al ways the best choice of each query, so we also leave the choice of memory allocation to the programmer (we pro- vide both row- and column-oriented options). Given these two inputs from the programmer, the framework then automatically allocates the memory in the form optimized for both streaming and compression, and performs execution on compressed data where applicable.

Our original intent was to perform our optimizations on top of existing state-of-the-art frameworks such as Trill, but as we show in Section 2, these frameworks are not properly tuned to exploit the full potential of existing memory subsystems. While we agree that their generality and ease of programming makes them a desirable choice in many situations, but we also envision TerseCades being further extended to be as general as these frameworks without sacrificing its performance.

4 Methodology and Microbenchmark

In this section, we will provide the detailed performance analysis of several key microbenchmarks and

[5] This idea has some similarity with the execution on encrypted data in homomorphic encryption [32], however in our case it is possible get performance even better than the uncompressed baseline.

demonstrate how different optimizations proposed in Section 3.3 affect their performance.

4.1 Methodology

In our experiments, we use two different CPU configurations. The first is a 24-core system based on Intel Xeon CPU E5-2673, 2.40GHz with SMT-enabled, and 128GB of memory, which is used in all microbenchmarks studies to have enough threads to put reasonable pressure on the memory subsystem. The second is a 4-core system based on Intel Xeon CPU E5-1620, 3.50GHz, SMT-enabled, and 16GB of memory. This system is used in all real applications experiments as it has better single-thread performance (especially higher per thread memory bandwidth). For our GPU experiments, we use NVIDIA GeForce GTX 1080 Ti with 11GB of GDDR5X memory. For FPGA prototyping, we use Altera Stratix V FPGA, 200MHz. In our evaluation, we use real applications scenarios (§5) and microbenchmarks from the STREAM suite [13]. We use *Throughput* (Millions of elements per second) and *Latency* (milliseconds) to evaluate streaming system performance; and *Compression Ratio* defined as uncompressed size divided by compressed size as the key metric for compression effectiveness.

4.2 Microbenchmark and Optimizations

SIMD-based Acceleration. To realize the full potential of data compression, it is critical to minimize the overhead due to data compression and especially decompression (that can be called multiple times on the same data). Luckily, the simplicity and inherent parallelism of the Base-Delta encoding algorithm allow to use SIMD vector instructions (e.g., Intel AVX) to perform multiple compressions/decompressions per instruction. Figure 6 shows the result of this optimization for *Add* benchmark from the STREAM benchmarks. We make two key observations from this figure.

First, when the number of threads is relatively small, this benchmark is more compute than memory limited. Hence reducing the computational overhead allows the *CharCompr.+V* version (compression plus vectorization) to almost completely match the ideal compression version (*Char*).[6] Second, when the number of threads increases (from 16 to 36), the additional overhead due to compression associated metadata becomes more important, and eventually when memory bandwidth becomes the only bottleneck, vectorization is not as useful in reducing the overhead anymore.

GPU/FPGA-based Acceleration. There are other hardware accelerators that can perform compression/decompression for Base-Delta encoding efficiently. For example, modern GPUs are suitable for massively parallel computations. We implemented the code that can perform multiple decompression operations in parallel using CUDA 8.0 [49] and tested this implementation using GeForce GTX 1080 Ti Graphics Card. Our results show that we can perform more than 32 Billion decompressions per second that is sufficient to satisfy the decompression rates required in realistic applications we will explore in Section 5. Note that this massive compute capability is frequently limited by the PCIe bandwidth that for our system was usually around 5-6 GB/sec.

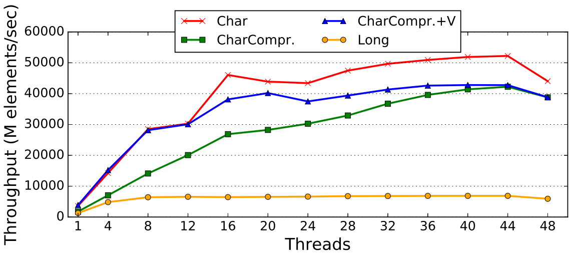


Figure 6: `Add` results with vectorization added.

Another option we explore is FPGA. We used SystemVerilog to implement the decompression logic and were able to run decompression at 200 MHz on a Stratix V FPGA board. We are able perform up to 744 Billion decompressions per second using this FPGA. Unfortunately, the bandwidth available through the PCIe again becomes the critical bottleneck limiting the number of decompressions we can perform. Nevertheless, it is clear that both GPUs and FPGAs can be efficiently used to hide some of the major data compression overheads.

Execution on Compressed Data. As we discussed in Section 3.3, the fact that the data is compressed usually comes with the burden of decompressing it, but it does not always have to be this way. There are several common scenarios when compressed data with Base-Delta encoding, can allow us to not only avoid decompression, but even execute the code faster. To demonstrate that, we take one benchmark called *Search* that essentially performs an array-wide search of a particular value (mimicking a very common *Where* operator in streaming). As we described in Section 3.3, when the data is represented in Base-Delta encoding, we take advantage of this fact and either completely avoid per value comparison within a batch (if the searched value is outside of the value range for this batch) or perform much more narrow 1-byte comparisons (8× less than in the original case).

Figure 7 presents the results of this experiment where *Compr.+Direct* is the mechanism that corresponds to compression with Base-Delta encoding and direct execution on compressed data as described in Section 3.3. Our key observation from this graph is that direct execution can not only dramatically boost the performance by

[6] Some additional overhead such as metadata and base storage overhead does not play a significant role here.

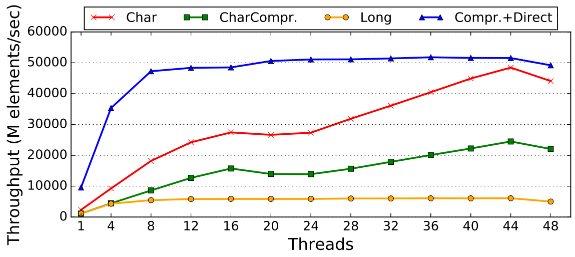


Figure 7: `Search` results with Direct Execution.

TimeStamp (8, **BD**)	ErrorCode (4, **EN+BD**)
SrcCluster (4, **HS+BD**)	DstCluster (4, **HS+BD**)
RoundTripTime(RTT) (4, **BD**)	

Table 1: Pingmesh record fields. Numbers in parenthesis are the bytes used to represent the field while letters are the compression algorithms we apply for that field. **BD**: base+delta; **HS**: string hashing; **EN**: enumeration.

avoiding the overhead of decompression (this is the performance gap between *Char*, ideal 8× compression with no overhead, and *CharCompr.*), but also significantly outperform the ideal compression case of *Char* (up to 4.1×). Moreover, it can reach almost the peak performance at just 8 threads, at which point it becomes fully memory bottlenecked, in contract to other cases where the peak performance is not reached until 44 threads are used. In summary, we conclude that direct execution on compressed data is a very powerful optimization that, when applicable, can by itself provide the relative performance benefits higher than that from data compression.

5 Applications

5.1 Monitoring and Troubleshooting

Pingmesh Data. Pingmesh [35] lets the servers ping each other to measure the latency and reachability of the data center network. Each measured record contains the following fields: timestamp, source IP address, source cluster ID, destination IP address, destination cluster ID, round trip time, and error code. Table 1 shows several of the fields that will be used in the queries in this paper. The measured records are then collected and stored. Data analysis is performed for dashboard reporting (e.g., the 50^{th} and 99^{th} latency percentiles), and anomaly detection and alerting (e.g., increased latency, increased packet drop rate, top-of-rack (ToR) switch down).

Pingmesh Queries. Here we describe implementation of several real queries on the Pingmesh data.

The query C2cProbeCount counts the number of error probes for the cluster-to-cluster pairs that take longer than certain threshold:

```
C2cProbeCount = Stream
  .HopWindow(windowSize, period)
  .Where(e => e.errorCode != 0 && e.rtt >= 100)
  .GroupApply((e.srcCluster, e.dstCluster))
  .Aggregate(c => c.Count())
```

The T2tProbeCount query is similar to the previous one, but uses Join to count the number of error probes for the ToR-to-ToR pairs:

```
T2tProbeCount = Stream
  .HopWindow(windowSize, period)
  .Where(e => e.errorCode != 0 && e.rtt >= 100)
  .Join(m, e => e.srcIp, m => m.ipAddr,
       (e,m)=> {e, srcTor=m.torId})
  .Join(m, e => e.dstIp, m => m.ipAddr,
       (e,m)=> {e, dstTor=m.torId})
  .GroupApply((srcTor, dstTor))
  .Aggregate(c => c.Count())
```

In the query, *m* is a table which maps server IP address to its ToR switch ID.

Compression Ratio, Throughput and Latency. In our experiments, we compare different designs that employ various compression strategies and optimizations: (i) *No Compression*, baseline system with all first-order optimizations described in Section 3.1, (ii) *Lossless* compression mechanism that employs Base-Delta encoding with simple mechanisms such as hashing and enumeration, (iii) *LosslessOptimized* mechanism that combines lossless compression described above with the SIMD acceleration and direct execution on compressed data, (iv) *Lossy* compression mechanism that uses lossy version of Base-Delta encoding in the cases where precise values are not needed, (v) *LossyOptimized* mechanism that combines lossy compression with the two major optimizations described in (iii). In addition, we evaluate two other designs: *Trill* streaming engine as a backend and *NonOptimized* design where we use TerseCades without any of the proposed optimizations.

The average compression ratio for these designs is as follows: *Lossless** designs have an average compression ratio on 3.1×, *Lossy** designs – 5.3×, and all other designs have no compression benefits (as compression is not used). Figure 8 compares the throughput of all the designs. First, as expected, first-order optimizations are critical in getting most of the benefits of implementing more specialized streaming engine in C++, leading to performance improvement of 9.4× over *Trill* streaming engine. As we remove most of the redundant computational overheads from the critical path, the memory bandwidth becomes a new major bottleneck. The four designs with data compression support overcome this bottleneck that limits systems' throughput – 32.3 MElems/s (Millions of elements-records per second).

Second, both *Lossless* and *Lossy* compression can provide significant throughput benefits as they have high average compression ratios (3.1× and 5.3×, correspondingly). However the full potential of these mechanisms

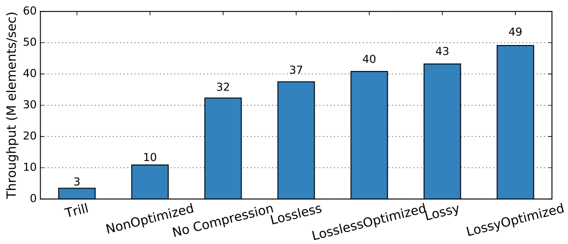


Figure 8: Throughput for the Pingmesh C2cProbeCount query. **Optimized** versions include both direct execution and SIMD optimizations.

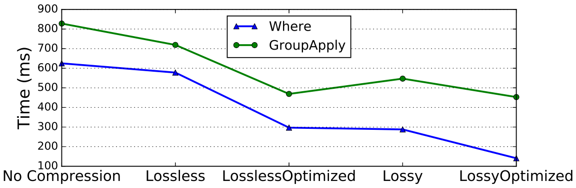


Figure 9: Execution times for Where and GroupApply operators used in the Pingmesh C2cProbeCount query. **Optimized** versions include direct execution and SIMD optimizations.

Mechanism	Throughput	Time
NoCompr.	27.7 MElems/s	2031 ms
LossyOptimized	38.3 MElems/s	1813 ms

Table 2: Throughput and Time (Join only) to perform the Pingmesh T2tProbeCount query that has Join operator.

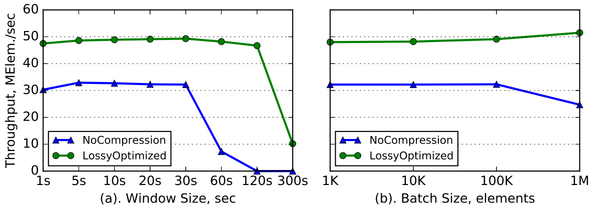


Figure 10: Throughput for the Pingmesh C2cProbeCount query with varying (a) window and (b) batch sizes.

is only uncovered when they are used in conjunction with vectorization (that reduces the overhead of compression/decompression) and direct execution on compressed data that for certain common scenarios (such as *Where* operator) can dramatically reduce the required computation (over the baseline). We conclude that efficient data compression through simple yet efficient compression algorithms **and** with proper optimizations can lead to dramatic improvement in streaming query throughput (e.g., 14.2× for troubleshooting query we analyzed).

Figure 9 shows similar results on the performance (execution time) of two major operators, *Where* and *GroupApply* for first five designs. Two observations are in order. First, both operators significantly reduce their execution time due to efficient usage of data compression, and the highest benefit is coming again from the most aggressive design, *LossyOptimized*. Second, the benefits due to compression and corresponding optimizations are more significant for *Where* operator – 4.6× improvement between *NoCompression* and *LossyOptimized* designs. This happens because *Where* operator benefits dramatically from the possibility of executing directly on compressed data (reducing the number of comparisons instructions needed to perform this operator).

Join Operator: Throughput and Execution Time. In order to demonstrate the generality of our approach, we also evaluate another scenario (T2tProbeCount query) that uses the *Join* operator. Table 2 shows the throughput and the execution time of this scenario for two designs: *NoCompression* and *LossyOptimized*. In this scenario, the throughput benefits are still significant (almost 1.4×), but the reduction in the operator's execution time is relatively small (around 12%). This is because most of the benefits are coming from the reduction in bandwidth consumption that happens mostly outside of *Join* – in *HopWindow* operator, while most of the computation performed to implement the *Join* operator is coming from hash table lookups.

Sensitivity to the Window Size. The window size used in the *HopWindow* operator can significantly affect the performance of the operators running after it. In order to understand if it is also a case for the troubleshooting scenario we consider, we study the performance of the two designs (*NoCompression* and *LossyOptimized*) on varying window size (from 1 second to 5 minutes).

Figure 10 (a) shows the results of this study. We make two observations. First, we observe that our proposed design, *LossyOptimized*, demonstrates stable throughput (around 50 MElems/sec) across different window sizes (from 1 to 120 seconds), with the variation below 5% in this range, and performance drops only at the 5-minute window. Second, in contrast to *LossyOptimized* design, the *NoCompression* design has a much shorter window of efficient operation (from 1 to 30 seconds).The primary reason for this is data compression that not only reduces the bandwidth consumed by the streaming engine, but also significantly reduces its memory footprint, allowing it to run on larger windows. Hence we conclude that compression allows to handle substantially larger windows (e.g., 4× larger) than the windows that can be efficiently handled without data compression.

Sensitivity to the Batch Size. In order to minimize the overhead of processing individual data elements, those elements are usually grouped by the streaming engines in batches. Hence the batch size becomes another knob to tune. We conduct an experiment where we vary the size

Mechanism	Throughput	Where	Group
NoCompr.	32.2 MElems/s	625 ms	828 ms
FrameRef.[33]	21.6 MElems/s	2453 ms	4078 ms
XPRESS [47]	8.5 MElems/s	7328 ms	13671 ms
LosslessOptimized	37.5 MElems/s	297 ms	469 ms
LossyOptimized	49.1 MElems/s	141 ms	453 ms

Table 3: Total throughput and time to perform Pingmesh query with optimized baseline without compression, two other compression algorithms, and our designs.

of the batch from 1K to 1M elements, and the results are shown in Figure 10 (b). We observe that in most cases the throughput of streaming engines are not very sensitive to the batch sizes in this range, except for one data point – the throughput of uncompressed design drop from 32.3 to 24.7 when going from 100K to 1M elements. Additional investigation shows that without compression for 1M elements the batch working set size exceeds the size of the available last level cache, limiting the benefits of temporal locality in the case of data reuse.

Sensitivity to the Compression Algorithms. In this work, we strongly argue that in order for compression algorithm to be applicable for streaming engine optimization, its complexity (compression/decompression overhead) should be extremely low and the algorithm itself has to be extensively optimized. We already showed that heavily optimized versions of arithmetic-based algorithm such as Base-Delta encoding can be efficient in providing significant performance benefits for streaming engines. We now compare our proposed designs with two well-known *lossless* compression algorithms used for in-memory data: FrameOfReference algorithm [33], arithmetic-based compression for low-dynamic range data, and XPRESS algorithm [47], dictionary-based algorithm that is based on LZ77 algorithm [68].

Our first comparison point is compression ratio, and as expected, both FrameOfReference and XPRESS outperform our *LosslessOptimized* algorithm in this aspect (compression ratios of 4.1× and 5.1×, respectively, vs. 3.1× for our design). However more importantly, as results in Table 3 indicate, both these algorithms prove not to be very practical for streaming engines, as their effect on throughput and execution time puts them below not only our proposed designs, but also significantly below uncompressed scenario. This happens because the cost of compression and decompression that are both on the critical path of execution outweighs the benefits of lower memory bandwidth consumption. We conclude that although it is important to have a compression algorithm with high compression ratio to provide reasonable performance improvements for streaming engines, it is even more critical to make sure those algorithms are efficient.

TimeStamp (8, **BD**)	Datacenter (3, **HS**)
Cluster (11, **HS**)	NodeId (10, **HS**)
VmId (36, **HS**)	CounterName (15, **EN**)
SampleCount (4, **BD**)	AverageValue (8, **ZFP**)
MinValue (8, **ZFP**)	MaxValue (8, **ZFP**)

Table 4: VM performance counter data fields. Numbers in parenthesis are the original sized of these fields, letters – compression algorithms used for them. **BD**: base+delta; **HS**: string hashing; **EN**: enum; **ZFP** [45].

5.2 IaaS VM Perf. Counter

Data. The cloud vendor regularly samples performance counters of an IaaS VM to determine a VM's health. If a VM is in an unhealthy state, recovery actions (e.g., relocate) will be taken to improve VM availability. Table 4 shows the fields for a performance counter record with the size to in-memory representations in the original analytics system. Each record contains the data source information (e.g., from which cluster, node and VM) and the actual values. At each regular interval, multiple such records of different types of counters will be emitted: e.g., CPU, network, disk. We use five datasets, i_1 to i_5, from different set of VMs in different timespan.

Queries. When processing these records, the data stream is grouped by timestamp and sources to get all the counters for a particular VM at each time point. We use a query to find the time and duration for a VM losing network activity: it first classifies the health of each perf counter group into different types (e.g., CPU active but network flat) and then scans the classified groups in ascending time for each unique VM to detect any type changes (e.g., from active to flat) and their durations.

Compressibility. Each performance counter record is represented with 111 bytes in memory in the original format, and a large portion of it can be compressed efficiently. For example, the `VmId` is a 36-character UUID string so that a VM can be universally uniquely identified across lifetime. But in the streaming scenario, in a given processing time window, the number of unique VMs tends not to be so large that they can be safely hashed to a 8-byte index. Note that absolute number of performance counters being emitted is large enough so that the hash table's overhead is amortized.

The compressibility can also come from batches of records rather than individual records. For example, the performance counter value is originally represented as a 8-byte double. The compressibility of a single record is not big (e.g., 2× if converted to integers). But efficient floating-point compression algorithm like ZFP can be applied across a stream of these counters to achieve high compression ratio. As Figure 11 shows, in certain runs, we can achieve near 6× compression ratio! The reason is that some VMs exhibit very regular performance patterns

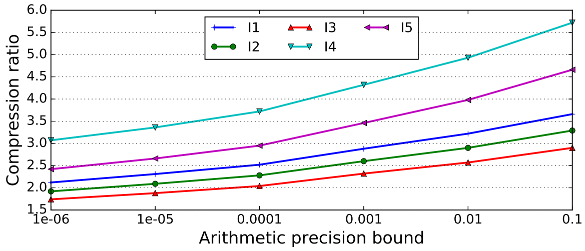


Figure 11: Compression ratio for various performance counters from VMs in a commercial cloud provider.

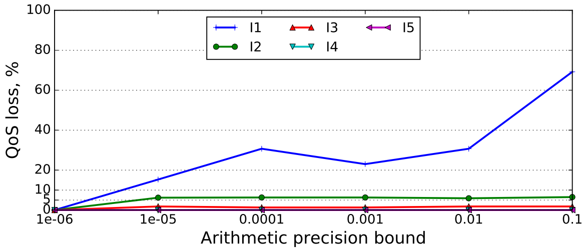


Figure 12: QoS metric loss for different compression ratios of performance counters.

(sometimes even constant) that can be exploited by ZFP. Of course, there are also some VMs that exhibit highly variable patterns, which will yield lower compression ratios (2.5× to 3× seen in our experiments).

Quality of Service. Algorithms like ZFP on floating-point values are lossy that can lose precisions when decompressed. Depending on the queries, the precision loss might sacrifice the QoS. This creates a trade-off between the compression ratio and the QoS level. We evaluated this trade-off using several queries on a set of real data coming from different regions' VMs in different timespan. The QoS loss metric is defined by the differences between the originally detected performance drop sessions with the new drop sessions: e.g., 1 additional session or missing session for a originally 100-session result is a 1% QoS loss. Figure 12 shows the result. We can see that for most datasets, the QoS loss level is low (below 5%), meaning that even aggressive lossy compression might be adopted without sacrificing QoS. For certain dataset, there is a high penalty (20% and more) because the absolute number of drop sessions are small.

5.3 IoT Data

Geolocation Data. This dataset contains GPS coordinates from the GeoLife project [2] by 182 users in a period of over three years. Figure 13 shows the average compression ratio of the dataset by using compression algorithms listed in Table 5. As we can see, these sensor data have significant redundancies because user movements tend not to have drastic changes. Even with an error bound of 10^{-6}, we can still achieve more than 4.5× compression ratio on average, creating significant opportunity for efficient real-time analytics over IoT data.

Latitude (8, **ZFP**)	Longitude (8, **ZFP**)
Altitude (4, **BD**)	TimeStamp (8, **BD**)

Table 5: Geolocation IoT data fields. Numbers and letters in parenthesis have the same meaning as Table 4.

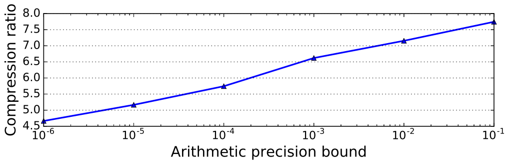


Figure 13: Compression ratio for GeoLocation IoT data.

Weather Data. We use another type of IoT data, 18,832,041 observations of weather data generated by various sensors during Hurricane Katrina in 2005 [3]. The measurements are stored as floating points in the data files but most of them are essentially integers due to the limited precisions of the sensors. Some metrics have a fixed number of digits after decimal points. They can be converted to integers as well to use integer compression algorithms like Base-Delta encoding. Across 18 metrics in the dataset, we can obtain an average of 3-4× compression ratios for each metric.

5.4 Real-World Implementation

We have built a distributed streaming system in our production data centers and implemented Pingmesh [35] usingTerseCades. In the original system, we used Trill [21] for streaming processing, which is now completely replaced with our new design. The whole system is composed of 16 servers each with two Xeon E5-2673 CPUs and 128G DRAM, running Windows Server 2016. Every server runs a frontend service and a backend service. The frontend services receive real-time Pingmesh data from a Virtual IP behind a load-balancer, and then partition the data based on the geo-region ID of the data and shuffle the data to the backend.

The whole system is designed to be fault tolerant – works well even if half services are down. The operations we perform are latency heatmap calculation at the 50^{th} and the 99^{th} percentiles, and several anomaly detections including ToR (Top of Rack) switch down detection. The aggregated Pingmesh input streaming is 2+ Gb/s, making it bandwidth-sensitive when properly optimized. The busiest server needs to process 0.5 millions events per second and uses 50% CPU cycles and 35GB memory. Using TerseCades to replace Trill, we reduce the sixteen servers to only one.

6 Related Work

Streaming System. Numerous streaming systems have been developed in both industry and literature [7, 18, 22, 41, 1, 10, 11, 66, 48, 56, 57, 65] to address the various needs for streaming processing: to name a few, Spark Streaming [66] applies stateless short-task batch computation to improve fault tolerance, MillWheel [10] supports user-defined directed graph of computation, Naiad [48] provides the timely dataflow abstraction to enable efficient coordination. One common requirement for these systems is to handle massive amount of unbounded data with redundancies. This motivates us to look into efficient data compression support in stream processing. Complimentary to these work, which focuses on high-level programming models in distributed environment, we focus on data compression in lower level core streaming engines that can potentially benefit these systems regardless of their high-level abstractions.

Memory Compression: Software and Hardware Approaches. Several mechanisms were proposed to perform memory compression in software (e.g., in the compiler [43] or the operating system [62]). While these techniques can be quite efficient in reducing applications' memory footprint, their major limitation is slow (usually software-based) decompression. This limits these mechanisms to compressing only "cold" pages (e.g., swap pages). Hardware-based data compression received some attention in the past [64, 8, 24, 30, 52]. However, proposed general-purpose designs had limited practical use either due to unacceptable compression/decompression latency or high design complexity, or because they require non-trivial changes to existing operating systems and memory architecture design.

Compression in Databases and Data Stores. Compression has been widely used to improve performance of databases [34, 58, 38, 25, 69, 5, 51, 46] and recent data stores [23, 9, 40] usually by trading-off overhead due to decompression for improved I/O performance and buffer hit rate. Some recent work investigates compression in the context of column-oriented databases [5, 6]that makes a few similar observations to our work: (i) adjacent entries in a column are often similar, (which helps improving compressibility), and (ii) some operators can run directly on compressed data to mitigate decompression costs (e.g., SUM aggregate on a run-length encoded column). The key difference in our work is that we apply compression in the streaming setting, and this puts significant limitations on the compression algorithm used (compared to offline data processing where latency is way less critical) that is now on the critical execution path. The proper choice of compression algorithm for streaming, reducing the key overheads of compression by using hardware acceleration, and using direct execution on compressed data (which not only avoids decompression, but actually executes faster than the baseline) are key contributions of our work that distinguish TerseCades from prior work on database compression.

One recent work, Succinct [9], supports queries that execute directly on compressed textual data (without indexes), significantly improving both memory efficiency and latency, but at the cost of complete redesign of how the data is stored in the memory. This is complementary to our work as our primary target is machine-generated, numerical data sets that proved to be more dominant in the streaming scenarios compared to textual data.

7 Conclusion

TerseCades is the first that attempts to answer the question of "Can data compression be effective in stream processing?". The design and optimizations of TerseCades answer these questions affirmatively. Our thorough studies and extensive evaluations using real stream workloads on production data further shed light on when and why compression might not be effective, as well as what can be done to make it effective. While our current implementation supports only a subset of operators supported by mature frameworks like Trill, we hope that by demonstrating the feasibility of data compression efficiency for streaming we will open the door for incorporating data compression in the next generation of stream processing engines.

References

[1] Apache Storm. http://storm.apache.org.

[2] GeoLife: Building social networks using human location history. https://www.microsoft.com/en-us/research/project/geolife.

[3] Linked sensor data. http://wiki.knoesis.org/index.php/LinkedSensorData.

[4] Aarnio, T., Brunelli, C., and Viitanen, T. Efficient floating-point texture decompression. In *System on Chip (SoC), 2010 International Symposium on* (2010).

[5] Abadi, D., Madden, S., and Ferreira, M. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (2006), SIGMOD '06, pp. 671–682.

[6] Abadi, D. J. *Query execution in column-oriented database systems*. PhD thesis, Massachusetts Institute of Technology, 2008.

[7] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. B. Aurora: A new model and architecture for data stream management. *VLDB J. 12*, 2 (2003), 120–139.

[8] Abali, B., Franke, H., Poff, D. E., Jr., R. A. S., Schulz, C. O., Herger, L. M., and Smith, T. B. Memory Expansion Technology (MXT): Software Support and Performance. *IBM J.R.D.* (2001).

[9] Agarwal, R., Khandelwal, A., and Stoica, I. Succinct: Enabling queries on compressed data. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (2015), NSDI'15, pp. 337–350.

[10] Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. MillWheel: Fault-tolerant stream processing at Internet scale. *PVLDB 6*, 11 (2013), 1033–1044.

[11] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1792–1803.

[12] Aws global infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/, 2017. [Online; accessed 16-April-2017].

[13] Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J. STREAM: The Stanford stream data manager. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003* (2003), p. 665.

[14] Arelakis, A., Dahlgren, F., and Stenstrom, P. Hy-Comp: A Hybrid Cache Compression Method for Selection of Data-type-specific Compression Methods. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015), MICRO-48.

[15] Arelakis, A., and Stenstrom, P. SC2: A Statistical Compression Cache Scheme. In *ISCA* (2014).

[16] Azure regions. https://azure.microsoft.com/en-us/regions/, 2017. [Online; accessed 16-April-2017].

[17] Bailis, P., Gan, E., Rong, K., and Suri, S. Prioritizing attention in fast data: Principles and promise. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data* (2017), SIGMOD '17.

[18] Barga, R. S., Goldstein, J., Ali, M. H., and Hong, M. Consistent streaming through time: A vision for event stream processing. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings* (2007), pp. 363–374.

[19] Burger, D., Goodman, J. R., and Kägi, A. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, USA, May 22-24, 1996* (1996), pp. 78–89.

[20] Burtscher, M., and Ratanaworabhan, P. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers 58*, 1 (Jan 2009), 18–31.

[21] Chandramouli, B., Goldstein, J., Barnett, M., DeLine, R., Platt, J. C., Terwilliger, J. F., and Wernsing, J. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB 8*, 4 (2014), 401–412.

[22] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F., and Shah, M. A. TelegraphCQ: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), ACM, pp. 668–668.

[23] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (2008), 4:1–4:26.

[24] Chen, X., Yang, L., Dick, R., Shang, L., and Lekatsas, H. C-pack: A high-performance microprocessor cache compression algorithm. *TVLSI* (2010).

[25] Chen, Z., Gehrke, J., and Korn, F. Query optimization in compressed database systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data* (2001), SIGMOD '01, pp. 271–282.

[26] Curial, S., Zhao, P., Amaral, J., Gao, Y., Cui, S., Silvera, R., and Archambault, R. On-the-fly structure splitting for heap objects. *ISMM* (2008).

[27] Deutsch, P. Gzip file format specification version 4.3. https://tools.ietf.org/html/rfc1952, 1996.

[28] Deutsch, P., and L. Gailly, J. Deflate compressed data format specification version 1.3, rfc. https://www.ietf.org/rfc/rfc1951.txt, 1996.

[29] Douglis, F. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *Winter USENIX Conference* (1993).

[30] Ekman, M., and Stenstrom, P. A Robust Main-Memory Compression Scheme. In *ISCA* (2005).

[31] Fowers, J., Kim, J.-Y., Burger, D., and Hauck, S. A scalable high-bandwidth architecture for lossless compression on fpgas. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2015), FCCM '15, IEEE Computer Society, pp. 52–59.

[32] Gentry, C., and Halevi, S. Implementing gentry's fully-homomorphic encryption scheme. Cryptology ePrint Archive, Report 2010/520, 2010. http://eprint.iacr.org/2010/520.

[33] Goldstein, J., Ramakrishnan, R., and Shaft, U. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering* (Feb 1998), pp. 370–379.

[34] Graefe, G., and Shapiro, L. D. Data compression and database performance. In *In Proc. ACM/IEEE-CS Symp. On Applied Computing* (1991), pp. 22–27.

[35] Guo, C., Yuan, L., Xiang, D., Dang, Y., Huang, R., Maltz, D., Liu, Z., Wang, V., Pang, B., Chen, H., Lin, Z.-W., and Kurien, V. Pingmesh: A large-scale system for data center network latency measurement and analysis. *SIGCOMM Comput. Commun. Rev. 45*, 4 (2015), 139–152.

[36] Hasselgren, J., and Akenine-Möller, T. Efficient depth buffer compression. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2006), GH '06.

[37] Huffman, D. A Method for the Construction of Minimum-Redundancy Codes. *IRE* (1952).

[38] Iyer, B. R., and Wilhite, D. Data compression support in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases* (1994), VLDB '94, pp. 695–704.

[39] JEDEC. DDR4 SDRAM Standard, 2012.

[40] Khandelwal, A., Agarwal, R., and Stoica, I. BlowFish: Dynamic storage-performance tradeoff in data stores. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (2016), NSDI'16, pp. 485–500.

[41] Kreps, J., Narkhede, N., and Rao, J. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece* (2011).

[42] Lattner, C., and Adve, V. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. *PLDI* (2005).

[43] Lattner, C., and Adve, V. Transparent Pointer Compression for Linked Data Structures. In *Proceedings of the ACM Workshop on Memory System Performance (MSP'05)* (2005).

[44] Lee, D., Ghose, S., Pekhimenko, G., Khan, S., and Mutlu, O. Simultaneous Multi Layer Access: A High Bandwidth and Low Cost 3D-Stacked Memory Interface. In *ACM TACO* (2015).

[45] Lindstrom, P. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (Dec 2014), 2674–2683.

[46] Microsoft. Data compression. https://docs.microsoft.com/en-us/sql/relational-databases/data-compression/, 2016.

[47] Microsoft. Ms-xca: Xpress compression algorithm. http://msdn.microsoft.com/enus/library/hh554002.aspx, 2016.

[48] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 439–455.

[49] NVIDIA. CUDA. https://developer.nvidia.com/cuda-toolkit, 2017.

[50] O'Neil, M. A., and Burtscher, M. Floating-point data compression at 75 gb/s on a gpu. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (New York, NY, USA, 2011), GPGPU-4, ACM, pp. 7:1–7:7.

[51] Oracle. Oracle advanced compression. http://www.oracle.com/technetwork/database/options/compression/overview/index.html, 2017.

[52] Pekhimenko, G., Seshadri, V., Kim, Y., Xin, H., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C. Linearly Compressed Pages: A Low Complexity, Low Latency Main Memory Compression Framework. In *MICRO* (2013).

[53] Pekhimenko, G., Seshadri, V., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches. In *PACT* (2012).

[54] Pool, J., Lastra, A., and Singh, M. Lossless Compression of Variable-precision Floating-point Buffers on GPUs. In *Interactive 3D Graphics and Games* (2012).

[55] Putnam, A., Caulfield, A., Chung, E., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Peterson, E., Smith, A., Thong, J., Xiao, P. Y., Burger, D., Larus, J., Gopal, G. P., and Pope, S. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)* (June 2014), IEEE Press, pp. 13–24.

[56] Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., and Zhang, Z. TimeStream: Reliable stream computation in the cloud. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013* (2013), pp. 1–14.

[57] Rabkin, A., Arye, M., Sen, S., Pai, V. S., and Freedman, M. J. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), pp. 275–288.

[58] Roth, M. A., and Van Horn, S. J. Database compression. *SIGMOD Rec. 22*, 3 (1993), 31–39.

[59] Ström, J., Wennersten, P., Rasmusson, J., Hasselgren, J., Munkberg, J., Clarberg, P., and Akenine-Möller, T. Floating-point Buffer Compression in a Unified Codec Architecture. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2008), GH '08.

[60] Sun, W., Lu, Y., Wu, F., and Li, S. DHTC: An Effective DXTC-based HDR Texture Compression Scheme. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2008), GH '08.

[61] Tremaine, R. B., Smith, T. B., Wazlowski, M., Har, D., Mak, K.-K., and Arramreddy, S. Pinnacle: IBM MXT in a Memory Controller Chip. *IEEE Micro* (2001).

[62] Wilson, P. R., Kaplan, S. F., and Smaragdakis, Y. The Case for Compressed Caching in Virtual Memory Systems. In *USENIX Annual Technical Conference* (1999).

[63] Woo, D. H., Seong, N. H., Lewis, D., and Lee, H.-H. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *HPCA* (2010).

[64] Yang, J., Zhang, Y., and Gupta, R. Frequent Value Compression in Data Caches. In *MICRO* (2000).

[65] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012* (2012), pp. 15–28.

[66] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. Discretized Streams: Fault-tolerant streaming computation at scale. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 423–438.

[67] Zhao, P., Cui, S., Gao, Y., Silvera, R., and Amaral, J. Forma: A framework for safe automatic array reshaping. *TOPLAS* (2007).

[68] Ziv, J., and Lempel, A. A Universal Algorithm for Sequential Data Compression. *IEEE TOIT* (1977).

[69] Zukowski, M., Heman, S., Nes, N., and Boncz, P. Superscalar ram-cpu cache compression. In *Proceedings of the 22Nd International Conference on Data Engineering* (2006), ICDE '06, pp. 59–.

Troubleshooting Transiently-Recurring Problems in Production Systems with Blame-Proportional Logging

Liang Luo[*], *Suman Nath*[†], *Lenin Ravindranath Sivalingam*[†], *Madan Musuvathi* [†], *Luis Ceze*[*]
[*]*University of Washington*, [†]*Microsoft Research*

Abstract

Many problems in production systems are *transiently recurring* — they occur rarely, but when they do, they recur for a short period of time. Troubleshooting these problems is hard as they are rare enough to be missed by sampling techniques, and traditional postmortem analyses of runtime logs suffers either from low-fidelity of logging too little or from the overhead of logging too much.

This paper proposes AUDIT, a system specifically designed for troubleshooting transiently-recurring problems in cloud-based production systems. The key idea is to use *lightweight triggers* to identify the first occurrence of a problem and then to use its recurrences to perform *blame-proportional logging*. When a problem occurs, AUDIT automatically assigns a *blame rank* to methods in the application based on their likelihood of being relevant to the root-cause of the problem. Then AUDIT enables heavy-weight logging on highly-ranked methods for a short period of time. Over a period of time, logs generated by a method is proportional to how often it is blamed for various misbehaviors, allowing developers to quickly find the root-cause of the problem.

We have implemented AUDIT for cloud applications. We describe how to utilize system events to efficiently implement lightweight triggers and blame ranking algorithm, with negligible to $< 1\%$ common-case runtime overheads on real applications. We evaluate AUDIT with five mature open source and commercial applications, for which AUDIT identified previously unknown issues causing slow responses, inconsistent outputs, and application crashes. All the issues were reported to developers, who have acknowledged or fixed them.

1 Introduction

Modern cloud applications are complex. Despite tremendous efforts on pre-production testing, it is common for applications to misbehave in production. Such misbehaviors range from failing to meet throughput or latency SLAs, throwing unexpected exceptions, or even crashing. When such problems occur, developers and operators most commonly rely on various runtime logs to troubleshoot and diagnose the problems.

Unfortunately, runtime logging involves an inherent tradeoff between logging sufficient detail to root-cause problems and logging less for lower overhead (see for instance [1, 2, 3, 4, 5]). Our experiments show (§6) that even for web applications that are not compute intensive, logging parameters and return values of all methods can increase latency and decrease throughput by up to 7%. Moreover, determining what to log is made harder by the fact that modern cloud and web applications involve multiple software components owned by different software developer teams. As a result, most logs generated today are irrelevant when root-causing problems [6].

To solve this problem, we make an important observation that many misbehaviors in production systems are *transiently-recurring*. As many frequent problems are found and fixed during initial phases of testing and deployment, we expect many problems in production systems to be rare and transient (the rarity makes it challenging to troubleshoot using sampling techniques [1, 2]). However, when they occur, they recur for a short amount of time for a variety of reasons, e.g., the user retrying a problematic request or a load-balancer taking some time to route around a performance problem (§2.2).[1]

Contributions. In this paper, we utilize the recurrence of these misbehaviors and present the design and implementation of AUDIT (AUtomatic Drill-down with Dynamic Instrumentation and Triggers): a *blame-proportional logging* system for troubleshooting transiently-recurrent problems in production systems. The basic idea is as follows. AUDIT uses lightweight triggers to detect problems. When a problem occurs, AUDIT automatically assigns a *blame rank* to methods in the application based on their likelihood of being rel-

[1] A notable exception to transient-recurrence are *Heisenbugs* which occur due to thread-interleaving or timing issues. AUDIT is not designed to troubleshoot these problems.

evant to the root-cause of the problem. Then AUDIT drills-down—it dynamically instruments highly-ranked methods to start heavy-weight logging on them until a user-specified amount of logs are collected. Over a period of time, logs generated by a method is proportional to how often the method is blamed for various misbehaviors and the overall logging is temporally correlated with the occurrence of misbehaviors. Developers analyze the logs *offline*, and thus AUDIT is complementary to existing techniques that help in interactive settings [7, 8].

We demonstrate the feasibility and benefits of AUDIT with the following contributions. First, AUDIT introduces *lightweight triggers* that continuously look for target misbehaviors. Developers can declaratively specify new triggers, describing target misbehaviors, the set of metrics to collect, and the duration for which to collect. The design of the trigger language is motivated by recent studies on misbehaving issues in production systems and when/where developers wish to log [3, 9, 10].

To evaluate the triggers and to blame-rank methods, AUDIT uses continuous end-to-end request tracing. To this end, our second contribution is *a novel tracing technique* for modern cloud applications built using Task Asynchronous Pattern (TAP), an increasingly popular way to write asynchrnous programs with sequential control flow and found in many languages including .NET languages, Java, JS/Node.js, Python, Scala, etc. AUDIT leverages system events at thread and method boundaries provided by existing TAP frameworks for monitoring and debugging purposes. AUDIT correlates these readily available events for lightweight end-to-end tracing. As a result, AUDIT introduces acceptable (from negligible to $< 1\%$) overhead in latency and throughput during normal operations. Note that AUDIT can also support non-TAP applications using known techniques based on instrumentation and metadata propagation [8, 11, 12] that are shown to have acceptable overheads in production systems.

Our third contribution is a novel ranking *algorithm that assigns blame scores to methods*. After a trigger fires, AUDIT uses the algorithm to identify high-ranked methods to initiate heavy-logging on them. AUDIT's blame ranking algorithm uses lessons from recent studies on where and what developers like to log for successful troubleshooting [3, 13]. It prioritizes methods where misbehavior originates (e.g., at a root exception that later causes a generic exception), that slow down requests, and that are causally related to misbehaving requests. It addresses key limitations of existing bottleneck analysis techniques that ignore critical path [14] or methods not on critical paths [15, 16, 17].

Our final contribution is an *evaluation* of AUDIT. We used AUDIT on a Microsoft production service and 4 popular open source applications. AUDIT uncovered previously-unseen issues in all the applications (§6.1). Many of the issues manifest only on production, as they are triggered based on user inputs and concurrency. All the issues have been reported to and acknowledged by developers of the applications. Some of them have already been fixed by developers with insights from logs generated by AUDIT.

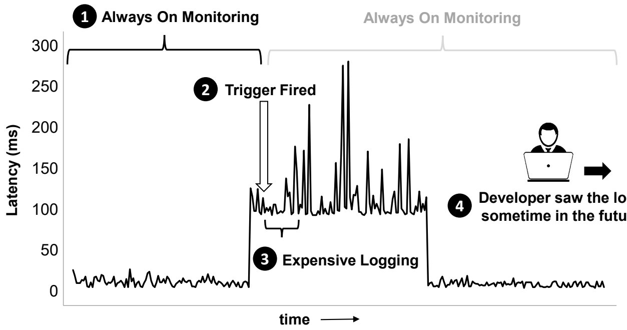


Figure 1: Timeline of AUDIT finding a performance bug in Forum.

2 Overview

2.1 A motivating case study

Microsoft Embedded Social. We start with an example demonstrating how AUDIT helped troubleshoot a problem in Microsoft Embedded Social (hereafter referred to as Social for brevity), a large-scale social service at Microsoft. Social is written in C#, deployed on Microsoft Azure, and is used by several applications and services in production. Social lets users add/like/search/delete posts, follow each other, and see feeds.[2]

Enabling AUDIT. AUDIT is easy to use. AUDIT works with unmodified application binaries and is enabled by simply setting a few environment variables. AUDIT comes with a set of triggers targeting common performance and exception-related problems. Social developers enabled AUDIT with these default triggers.

The problem. Figure 1 shows a performance problem that occurred in production: the latency of retrieving *global feeds* increased for a few hours. The developer was offline during the entire time and later relied on AUDIT logs to troubleshoot the issue.

AUDIT in operation. An AUDIT trigger fired shortly after the sudden spike in latency ((2) in Figure 1). For all misbehaving requests, AUDIT logged end-to-end *request trace* consisting of the request string, names and caller-callee relationship of executed methods. In addition, its blame ranking algorithm selected top-k (k = 5 by default) ranked methods and dynamically instrumented them to log their parameters and return values.

[2] Open source SDKs are available on GitHub, e.g., https://github.com/Microsoft/EmbeddedSocial-Java-API-Library

The *heavyweight logging* continues for a short time (5 minutes by default, stage (3) in Figure 1). This spatially- and temporally-selective logging helped the developer to root-cause of the problem, even long after the problem disappeared (stage (4) in Figure 1).

Troubleshooting with AUDIT logs. AUDIT's request traces showed that the misbehaving request was retrieving the global feed. The feed consists of a list of post-ids and contents; the latter is stored in a back-end store (Azure Table Storage) and is cached (in Redis) to achieve low-latency and high-throughput. Request tracing showed that the spike was due to post contents consistently missing the cache, albeit without revealing the cause of cache misses.

Among the methods AUDIT selected for heavyweight logging was a method that queries the backing store (Azure Table Storage). The logged arguments showed that the spike was caused by one particular post id, which according to logged return value didn't exist in the backing store. This inconsistency lead to the root-cause of the bug — a post id was present in the global feed but its contents were missing.

This inconsistency occurred when a user deleted a post. Social deleted its content from cache and backing store but failed to delete its id from the global feed due to a transient network failure. This inconsistency was not visible to users as missing posts are omitted in feeds, but it created a persistent performance spike. The inconsistency eventually disappeared when the problematic id was moved out of the global feed by other posts.

In addition to pinpointing the inconsistency, AUDIT also helped the developer root-cause the inconsistency to the failure in the delete operation through its detailed failure logging. We discuss the bug in more detail in §6.1.1. The developer chose to fix the problem by implementing negative caching, where Redis explicitly stores that a post is deleted from the backing store.

The case study demonstrates the value of AUDIT: it can capture *useful logs* for *relatively rare issues* that may appear in production or large-scale tests when the developer is absent. Moreover, logs are collected *only for a short period* after the issue occurs, reducing the log collection overhead and making it suitable even for expensive logging operations.

2.2 Transiently-recurring Problems

We argue that many problems in cloud-based production systems are transiently-recurring. First, if an error is frequent, it will most likely be detected and fixed during pre-production testing and staging of the application. Second, many problems are due to infrastructure problems such as transient network hardware issues [9, 18]; SLAs from cloud service providers ensure that such problems are rare and fixed within a short window of time. Therefore, cloud applications commonly use the "retry pattern" [19, 20] where the application transparently retries a failed operation to handle transient failures. Third, some problems are due to user inputs (e.g., malformed). Such errors are rare in well-tested production systems; however, once happened, they persist till the user gives up after several retries [21].

Note that AUDIT is also useful for troubleshooting errors that appear frequently—as long as they persist for a small window of time (e.g., not Heisenbugs).

3 AUDIT design

At a high level, AUDIT consists of four components: (1) declarative *triggers* for defining misbehaving conditions (§ 3.1), (2) a light-weight *always-on monitoring* component that continuously evaluates trigger conditions and collects request execution traces (§3.2 and § 4), (3) a *blame assignment algorithm* to rank methods based on their likelihood of being relevant to the root cause of a misbehavior (§ 3.3), and (4) a *selective logger* that uses dynamic instrumentation to enable and disable logging at top-blamed methods (§ 3.4).

3.1 AUDIT triggers

Trigger Language. AUDIT triggers are similar to *Event-Condition-Action* rules that are widely used in traditional databases [22] and in trigger-action programming such as IFTTT [23]. A key challenge in designing AUDIT's trigger language is to make it concise, yet expressive enough for a developer to specify interesting misbehaviors and useful logs. Before we elaborate on the rationale behind our choice, we first describe the four key components of an AUDIT trigger:

(1) `ON`. It specifies when (`RequestStart`, `RequestEnd`, `Exception`, or `Always`) the trigger is evaluated.

(2) `IF`. It describes a logical condition that is evaluated on the `ON` event. The condition consists of several useful properties of the request `r` or the exception `e` such as `r.Latency`, `e.Name`, `r.ResponseString`, `r.URL`, etc. It also supports several streaming aggregates: `r.AvgLatency(now,−1min)` is the average latency of request `r` in the last 1 min, `e.Count(now,−2min)` is the number of exception `e` in the last 2 mins, etc.

(3) `LOG`. It describes what to log when the `IF` condition satisfies. AUDIT supports logging `RequestActivity`[3] and `method` of a request. A key component of `LOG` is `ToLog`, which indicates target metrics to log: e.g., `args`, `retValue`, `exceptionName`, `latency`, `memoryDump`. Logs can be collected for requests matching (or not matching) the `IF` condition with a sampling probability of `MatchSamplingProb` (or `UnmatchSamplingProb`,

[3] A request activity graph (§3.3) consists of all methods invoked by the request as well as their causal relationship.

```
1 DEFINE TRIGGER T
2  ON RequestEnd R
3  IF R.URL LIKE 'http:*GetGlobalFeed*'
4     AND R.AvgLatency(-1min, now) > 2 * R.
          AvgLatency(-2min, -1min)
5  LOG RequsetActivity A, Top(5) Methods M
6     WITH M.ToLog=args, retValues
7     AND MatchSamplingProb = 1
8     AND UnmatchSamplingProb = 0.3
9  UNTIL (10 Match,10 Unmatch) OR 5 Minutes
```

Figure 2: An AUDIT trigger that fires when the latency of the global feed page in Social increases.

respectively). This enables comparing logs from "good" and "bad" requests. Finally, AUDIT supports logging all or a specified number of top performance-critical methods (with the Top() keyword). The later is useful when the request involves a large number of methods and instrumenting all of them would incur a high runtime overhead. Users can also define custom logging library that AUDIT can dynamically load and use.

(4) `UNTIL`**.** It describes how long or how many times the `LOG` action is performed.

Language Rationale. As mentioned, AUDIT's trigger language is motivated by prior works [3, 9, 10]. The general idea of enabling logging on specific misbehaving conditions (specified by `ON` and `IF`) and disabling it after some time (specified via `UNTIL`) addresses a key requirement highlighted in a recent survey of 54 experienced developers at Microsoft by Fu et. al [3]. The authors also analyzed two large production systems and identified three categories of *unexpected situation* logging. AUDIT's triggers support all of them: (1) exception logging, through `exceptionName` and `RequestActivity`, (2) return-value logging, via `retValue`, and (2) assertion-check logging, via `args`. The `ToLog` metrics are chosen to support common performance and reliability issues in production systems [9]. Logging both "good" and "bad" requests is inspired by statistical debugging techniques such as Holmes [10].

An Example Trigger. Figure 2 shows a trigger that can be used by Social for the scenario described in §2.1. The trigger fires when the average latency of the global feed page computed over a window of 1 minute increases significantly compared to the previous window. AUDIT starts logging all requests matching the `IF` condition and 30% of requests not matching the condition (for comparison) once the trigger fired. For each such request, AUDIT logs the *request activity*, consisting of all sync/async methods causally related to the request. Additionally, it assigns a blame rank to the methods and logs parameters and return values of 5 top-ranked methods. AUDIT continues logging for 10 matched and 10 unmatched requests, or for a maximum of 5 minutes.

Specifying Triggers. The trigger in Figure 2 may look overwhelming, with many predicates and parameters. We use this trigger for illustration purpose. In practice, a developer does not always need to specify all trigger parameters, letting AUDIT use their default values (all numerical values in Figure 2 are default values). Moreover, AUDIT comes with a set of predefined triggers that a developer can start with in order to catch exceptions and sudden spikes in latency and throughput. Over time, she can dynamically refine/remove existing triggers or install new triggers as she gains more operational insights. For example, the trigger in Figure 2 minus the predicate in Line 3 is a predefined trigger; Social developers modified its scope to global feed requests.

3.2 Always-on monitoring

AUDIT runtime continuously evaluates installed triggers. AUDIT instruments application binaries to get notified of triggering events such as exceptions, request start and end, etc. AUDIT automatically identifies instrumentation points for web and many cloud applications that have well-defined start and end methods for each request; AUDIT users can declaratively specify them for other types of applications. The handlers of the events track various request and exception properties supported by AUDIT trigger language. In addition, if needed by active triggers, AUDIT maintains lightweight streaming aggregates such as `Count`, `Sum`, and `AvgLatency` over a window of time.

In addition, AUDIT uses end-to-end causal tracing to continuously track identity and caller-callee relationships of methods executed by each request. For general applications, AUDIT uses existing tracing techniques based on instrumentation and metadata propagation [1, 8, 24, 25, 26, 27, 28]. For cloud applications using increasingly popular Task Asynchrnous Pattern (TAP), AUDIT uses a more lightweight and novel technique that we describe in §4.

AUDIT represents causal relationships of methods with a *request activity graph* (RAG), where nodes represent instances of executed methods and (synchronous or asynchrnous) edges represent caller-callee relationships of the nodes. A *call chain* to a node is the path from the root node to that node. (A call chain is analogous to a stack trace, except that it may contain methods from different threads and already completed methods.)

For multi-threaded applications, a RAG can contain two special types of nodes. A *fork node* invokes multiple asynchronous methods in parallel. A *join node* awaits and starts only after completion of the its nodes. A join node is an *all-join* node (or, *any-join* node), if it waits for all (or, any, respectively) of its parents node to complete. For each method in the RAG, AUDIT also tracks four timestamps: a (t_{start}, t_{end}) pair indicating when the method starts and ends, and a $(t_{pwStart}, t_{pwEnd})$ pair indicating when the method's parent method starts and ends waiting for it to complete (more details in §4).

3.3 Blame assignment and ranking

After a misbehaving request fires a trigger, AUDIT uses a novel algorithm that ranks methods based on their blames for a misbehavior – the higher the blame of a method, the more likely it is responsible for the misbehavior. Thus, investigating the methods with higher blames are more likely to be helpful in troubleshooting the misbehavior.

To assign blames, AUDIT relies on RAGs and call chains of misbehaving requests, as tracked by the always-on monitoring component of AUDIT.

3.3.1 Exception-related triggers

On an exception-related trigger, AUDIT uses the call chain ending at the exception site to rank methods (on the same or different threads). Methods on the call chain are ranked based on their distance from the exception – the method that throws the exception has the highest rank and methods nearer to the exception are likely to contain more relevant information to troubleshoot root causes of the exception (as suggested by the survey in [3]).

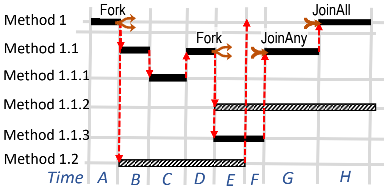
3.3.2 Performance-related triggers

On a performance-related trigger, AUDIT uses a novel bottleneck analysis technique on the RAGs of misbehaving requests. Existing critical path-based techniques (e.g., Slack [15], Logical Zeroing [16], virtual speedup [17]) fall short of our purpose because they ignore methods that should be logged but are not on a critical path or have very little exclusive run time on critical path. Techniques that ignore critical paths (e.g., NPT [14]) also miss critical methods that developers wish to log. §6 shows several real-world examples that illustrate these limitations.

Blame assignment. AUDIT addresses the above limitations with a new metric called *critical blame* that combines critical path, execution time distribution, and join-node types. Given a RAG, computation of critical blames of methods consists of two steps.

First, AUDIT identifies critical paths in the RAG. A critical path is computed recursively, starting from the last node of the RAG. Critical path to a node includes the node itself and (recursively computed) critical paths of (1) all parent non-join nodes, (2) longest parents of all-join nodes, and (3) shortest parents of any-join nodes. Each method in the critical path has the property that if its runs faster, total request latency goes down. See §5 for how these timestamps are derived.

Second, AUDIT assigns to each method on the critical path a *critical blame* score, a metric inspired by NPT[14]. Critical blame for a method consists of its exclusive and fair share of time on the critical path. Figure 3 illustrates how AUDIT computes critical blames of various methods in a RAG. Recall that each node in the RAG has four timestamps: a (t_{start}, t_{end}) pair and a $(t_{pwStart}, t_{pwEnd})$ pair. At a given time t, we consider a node to be *active* if t is within its t_{start} and t_{end} but not within any of its child method's $t_{pwStart}$ and t_{pwEnd}.



Method	Blame
1	A+H/2
1.1	(B+D+G)/2
1.1.1	C/2
1.1.2	E/3+(F+G+H)/2
1.1.3	E/3+F/2
1.2	(B+C+D)/2+E/3

Figure 3: Critical blame assignment to methods. Solid edges represent methods on the critical path.

To compute critical blames of methods, AUDIT linearly scans the above timestamps of all methods (including the ones not in the critical path) in increasing order. Conceptually, this partitions the total request lifetime into a number of discrete *segments*, where each segment is bounded by two timestamps. In Figure 3, the segments are marked as $A, B, \ldots$ at the bottom. At each segment, AUDIT distributes the total duration of the segment to all methods active in that segment. For example, in the segment A, Method 1 is the only active method, and hence it gets the entire blame A. In segment B, methods 1.1 and 1.2 are active, and hence they both get a blame of $B/2$. Total blame of a method is the sum of all blames it gets in all segments (Method 1's total blame is $A + H/2$).

Selecting top methods. Given a target number n, AUDIT first selects the set $\mathbf{B}_1$ of n highest-blamed methods on the critical path. Let α be the lowest blame of methods in $\mathbf{B}_1$. AUDIT then compute another set $\mathbf{B}_2$ of methods not in the critical path whose execution times overlap with a method in $\mathbf{B}_1$, and whose blame scores are $\geq \alpha$. Finally, AUDIT computes $\mathbf{B} = \mathbf{B}_1 \cup \mathbf{B}_2$, and outputs all unique method names in $\mathbf{B}$. Essentially, the algorithm includes all slow critical methods and some slow non-critical methods that interfere with the critical methods.

Note that size of $\mathbf{B}$ can be larger (as it takes non-critical methods in $\mathbf{B}_2$) or smaller (as it ignores method instances) than n. If needed, AUDIT can try different sizes of $\mathbf{B}_1$ to produce a $\mathbf{B}$ whose size is close to n.

The intuition behind the above algorithm is as follow: (1) we want to blame only tasks that are actually running for the time they use; (2) we want co-running tasks to share the blame for a specific time period, assuming fixed amount of resources; (3) we want to first focus on tasks that are critical path as they affect runtime directly and (4) we want to include selective non-critical path tasks as they can be on the next longest path, may interfere with tasks on the critical path, and not all critical path methods can be modified to run faster. §6.2 compares critical blame to other metrics quantitatively.

3.4 Enabling and disabling logging

AUDIT uses dynamic instrumentation to temporarily inject logging statements into blamed methods. The process works with unmodified applications and only requires setting few environment variables pointing to AUDIT library. Like Fay [7] and SystemTap [29], AUDIT supports instrumenting tracepoints at the entry, normal return, and exceptional exit of any methods running in the same address space as the application.

Specifically, AUDIT decorates each selected method with three callbacks. `OnBegin` is called as the first instruction of the method, with the current object and all arguments. It returns a local context that can be correlated at two other callbacks: `OnException`, called with the exception object, and `OnEnd`, called with the return value. These callbacks enable AUDIT to collect a variety of drilldown information. To log method parameters, global variables, or system parameters such as CPU usage, AUDIT uses `OnBegin`. To log return values, it uses `OnEnd`. Latency of a method is computed by taking timestamps at `OnBegin` and `OnEnd`. To collect memory dumps on exception, AUDIT uses `OnException`.

4 Optimizations for TAP applications

Task asynchronous pattern (TAP) is an increasingly popular programming pattern[4], especially in cloud applications that are typically async-heavy. Unlike traditional callback-based Asynchronous Programming Model (APM), TAP lets developer write non-blocking asynchronous programs using a syntax resembling synchronous programs. For example, TAP async functions can return values or throw exceptions to be used or caught by callers. This makes TAP intuitive and easier to debug, avoiding callback hell [30]. Major languages including .NET languages (C#, F#, VB), Java, Python, JavaScript, and Scala support TAP. In Microsoft Azure, for many services, TAP is provided as the *only* mechanism to do asynchronous I/O. Amazon AWS also provides TAP APIs for Java [31] and .NET [32].

One contribution of AUDIT is to show that for TAP applications, it is possible to construct RAG and call chains extremely efficiently, without extensive instrumentation or metadata propagation. Our techniques provide *intra-machine* RAG and call chains, where APIs of nodes may cross machine boundaries but edges are within the same machine. We focus only on such RAGs as we found them sufficient for our target cloud applications; if needed, inter-machine edges can be tracked by using the techniques used by Pivot Tracing [8].

4.1 Continuous tracking of RAGs

AUDIT utilizes *async lifecycle events* provided by existing TAP frameworks for constructing RAGs. For debugging and profiling purpose, all existing TAP frameworks we know provide light-weight events or mechanisms indicating various stages of execution of async methods. Examples include ETW events in .NET [33], AsyncHooks [34] in Node.js, Decorators for Python AsyncIO [35], and RxJava Plugin [36] for Java. The events provide limited information about execution times and caller-callee relationships between some async methods, based on which AUDIT can construct RAGs. Using lifecycle events for tracing is not trivial. Depending on the platform, the lifecycle events may not directly provide all the information required to construct a RAG. We describe a concrete implementation for .NET in § 5.

4.2 On-demand construction of call chains

Even though call chain is a path in the RAG, AUDIT uses a separate mechanism to trace it for TAP applications. The advantage is that it lazily constructs a call chain on-demand, only *after* an exception-related trigger fires. Thus, the mechanism has *zero cost* during normal execution, unlike existing proactive tracking techniques [11, 12, 37]. AUDIT combines several mechanisms to achieve this.

AUDIT exception handler. AUDIT registers AUDIT event handler (AEH) to system events that are raised on all application exceptions. Examples of such events are *First Chance Exception* [38] for .NET and C++ for Windows, *UncaughtExceptionHandler* [39, 40] for Java, and *RejectionHandled* [41] for JavaScript.

AUDIT's exception tracing starts whenever the application throws an exception that satisfies a trigger condition. Consider `foo` synchronously calling `bar`, which throws an exception. This will invoke AEH with `bar` as the call site and a stacktrace at AEH will contain `foo`. This enables AUDIT to infer the RAG edge from `foo` to `bar`. If, however, `bar` runs asynchronously and in a different thread than `foo`, stacktrace won't contain `foo`. To infer the async edge from `foo` to `bar`, AUDIT relies on how existing TAP frameworks handle exceptions.

Exception propagation in TAP. Recall that TAP allows an async method to throw an exception that can be caught at its caller method. When an exception `e` is thrown in the async method `bar`, the framework first handles it and then revisits or rethrows *the same* exception object `e` when the caller method `foo` retrieves the result of

[4] To quantify TAP's popularity, we statically analyzed all C# (total 18K), JavaScript (Node.js) (16K), and Java (Android) (15K) GitHub repositories created between 1/1/2017 and 6/30/2017. Our conservative analysis, which may miss applications using 3rd party TAP libraries, identified 52% of C#, 50% of JavaScript, and 15% of Java projects using TAP. The fractions are significantly higher than the previous year (e.g., 35% higher for C#), showing increasing popularity of TAP.

bar [42]. This action may trigger another first chance exception, calling the AEH with e.

AUDIT correlates on exception objects to discover async caller methods in a call chain and uses the order in which the AEHs are invoked in various methods to establish their order. In general, a call chain may contain a combination of synchronous and asynchronous edges. AUDIT uses stack traces to find small chains of consecutive synchronous edges, and correlates on exception objects to stitch the chains.

An application may catch one exception e1 and rethrow another exception e2. This pattern is dominant especially in middleware, where library developers hide low-level implementation details and expose higher level exceptions and error messages. The exception tracing technique described so far will produce two separate call chains, one for e1 and another for e2. However, since e1 has triggered e2, causally connecting the two chains can be useful for troubleshooting and root cause analysis [3].

Inheritable thread-local storage (ITS). AUDIT uses ITS to connect correlated exceptions. Inheritable thread-local storage allows storing thread-local contents that automatically propagate from a thread to its child threads. This is supported in Java (InheritableThreadLocal), .NET (LogicalCallContext), and Python (AsyncIO Task Local Storage[43]). Using ITS is expensive due to serialization and deserialization of data at thread boundaries. Existing causal tracing techniques use ITS all the time [27]; in contrast, AUDIT uses it only for exception tracing and on demand.

When e1 and e2 happens in the same thread, AUDIT can easily correlate them by storing a correlation id at the AEH of e1, and then using the id at the AEH of e2.

If e2, however, is thrown on a different thread than e1, the situation is more subtle. This is because e2 is thrown on the parent (or an ancestor) of e1's thread, and the correlation id stored in a thread's ITS is not copied *backward* to the parent thread's context (it is only copied forward to child threads).

To address this, AUDIT combines ITS with how TAP propagates exceptions across threads (described above). More specifically, AUDIT uses the first exception e1 as the correlation id and relies on TAP to propagate the id to the parent thread, which can correlate it to e1. The AEH for e2 stores e1 in ITS for further correlating it with other related exceptions on the same thread.

5 Implementation

We here describe our implementation of AUDIT for TAP applications written in .NET for Windows and cross-platform .NET Core.

Listening to exceptions. AUDIT listens to AppDomain.FirstChanceException to inspect all exceptions thrown by the application. First chance exception is a universal debugging concept (e.g., catch point in GDB, first chance exception in Visual Studio). A first chance exception notification is raised as soon as a runtime exception occurs, irrespective of whether it is later handled by the application.

Request tracing. For efficiently constructing the RAG of a request, AUDIT uses TplEtwProvider, an ETW-based [33] low overhead event logging infrastructure in .NET. TplEtwProvider generates events for the lifecycle of tasks in TAP.

Specifically, AUDIT uses TraceOperationBegin event to retrieve the name of a task. TaskWaitBegin is used for timestamp when a parent task transitions to suspended state and starts to wait on a child task. TraceOperationRelation is used to retrieve children tasks of a special join task (WhenAll, WhenAny), these join tasks are implemented in a special way such that they do not produce other life cycle events. At last, TraceOperationComplete, TaskWaitEnd, TaskCompleted, RunningContinuation, TaskWaitContinuationComplete are used to track the completion of a task. Many events are used because not all tasks generate the same event.

Constructing RAG based only on TPL ETW events is challenging for two key reasons, which AUDIT addresses by utilizing semantics of the events. First, ETW events are not timestamped by their source, but by the ETW framework after it receives the event. The timestamps are not accurate representation of the event generation times as the delivery from source to ETW framework can be delayed or out-of-order. To improve the quality of timestamps, for each method on the RAG, AUDIT aggregates multiple ETW events. For example, ideally, the t_{end} timestamp should come from the TaskCompleted ETW event. However, TPL generates other events immediately after a task completes. AUDIT takes the earliest of the timestamps of any and all of these events, to tolerate loss and delayed delivery of some events. AUDIT also uses the fact that in a method's lifetime, $t_{start} \geq t_{pwStart} \geq t_{end} \geq t_{pwEnd}$. Thus, if, e.g., all ETW events related to t_{start} are lost, it is set to $t_{pwStart}$.

Second, TPL does not produce any ETW events for join tasks, which are important parts of RAG. AUDIT uses reflection on the joining tasks (that produce ETW events) to identify join tasks, as well as their types (all-join or any-join). The t_{start} and t_{end} timestamps of a join task is assigned to the t_{start} and t_{end} timestamps of the shortest or the longest joining task, depending on whether the join task is any-join or all-join, respectively.

Dynamic instrumentation AUDIT uses .NET's profiling APIs to dynamically instrument target methods during runtime. The process is similar to dynamically instrumenting Java binaries [44].

6 Evaluation

We now present experimental results demonstrating:

1. AUDIT can effectively root-cause transiently recurring problems in production systems (§6.1).
2. AUDIT's blame ranking algorithm is more effective in root-causing than existing techniques (§6.2)
3. AUDIT has acceptably small runtime overhead for production systems, and its TAP-related optimizations further reduce the overhead (§6.3).

6.1 Effectiveness in root-causing bugs

We used AUDIT on five high-profile and mature .NET applications and identified root causes of several transiently recurring problems and bugs (Table 1). All the issues were previously unknown and are either acknowledged or fixed by developers. In all cases, AUDIT's ability to trigger heavyweight logging in a blame-proportional manner were essential to resolve problems.

6.1.1 Case study: Embedded Social

In § 2.1, we described one performance issue AUDIT found in Embedded Social (Social), a large-scale production social service in Microsoft. We now provide more details about Social and other issues AUDIT found in it. At the time of writing, Social had millions of users in production and beta deployments. We deployed Social in a deployment cluster. We enabled AUDIT with a generic exception trigger and a few performance triggers for latency-sensitive APIs (e.g. Figure 2).

Social 1: The persistent performance spike (Figure 1) arose because of an inconsistency caused by a failure (network timeout) during post deletion – the post id in the feed was left undeleted. Social swallowed the actual exception and produced only a high level exception for the entire delete operation. AUDIT logged the entire chain, pinpointed that post contents were deleted, but global feed deletion failed. AUDIT also logged the request URL, which identified the post id that was being deleted. The RAGs produced by the performance trigger showed the persistent store being consistently hit for one post. AUDIT's blame ranking algorithm top-ranked the persistent store query method, dynamically instrumented it, and logged arguments and return value for the next few requests to the global feed. The logged arguments showed the problematic post id causing the spike and the logged return value (`NULL`) indicated that it was deleted from the store and pointed to lack of negative caching as an issue. The post id matched the one logged during delete operation failure, which explained the bug.

Social 2: AUDIT revealed a few more transiently recurring issues related to lack of negative caching. For example, Social recommends a list of users with high follower count to follow. In the corner case of a popular user not following anyone, Social did not create an entity for the following count in the persistent store (and thus in the cache). In this case, the main page persistently missed the cache when reporting such users in the recommended list. AUDIT correctly assigned blame to the count-query method and logged both the user id (as part of parameters) and the return value of 0. Social's developers implemented negative caching to fix them.

Social 3: AUDIT's exception trigger in Social helped root-cause several transiently recurring request failures. We discuss a couple of them here. "Likes" for a post are aggregated and persisted to ATS using optimistic concurrency. When a specific post became hot, updates to ATS failed because of parallel requests. Through drill down, AUDIT pinpointed the post id (parameter) of the hot post and showed that like requests were failing only for that particular post id and succeeding for others.

Social 4: As posts are added, Social puts them in a queue and indexes the content of the posts in a backend worker. Typical to many systems, when a worker execution fails, the jobs are re-queued and retried a few times before being dead-lettered. This model perfectly fits AUDIT's triggered logging approach. After the first time a worker fails on a request, AUDIT triggers expensive parameter logging for subsequent retries. By logging their parameters, AUDIT root-caused many content-related bugs during indexing due to bad data formats.

We also found AUDIT useful in root-causing rare but recurrent problems in several open-source projects. Below we summarize the symptoms, AUDIT logs, and root cause of the problems.

6.1.2 Case study: MrCMS

MrCMS[45] is a content management system (CMS) based on the ASP.NET 5 MVC framework.

Symptoms. On a rare occasion, after an image is uploaded, the system crashed. Then the system became permanently unusable, even after restarting.

AUDIT logs. The AUDIT log from the first occurrence of the problem indicated an unhandled `PathTooLongException`. This was surprising because MrCMS checks for file name length. The methods on the call chain, however, indicated that the exception happened when MrCMS was creating thumbnail for the image. After AUDIT instrumented methods on the call chain, recurrence of the problem (i.e., recurrent crashing after restart) generated logs including method parameters. This included the actual file name for which a file system API was throwing the exception.

Root cause and fix. When image files are uploaded, MrCMS generates thumbnails with the image file name suffixed with dimensions. Thus, when an input file name is sufficiently long, the thumbnail file name can exceed the filesystem threshold which is unchecked and caused the crash. As most bugs in production systems, the fix for

Application	Issue	Root cause based on AUDIT log	Status from devs
Social 1	Performance spike when reading global feeds	Deleted operation failed to delete the post from global feeds	Fixed
Social 2	Poor performance reading user profiles with no following in “Popular users” feed	Lack of caching zero count value	Fixed
Social 3	Transient “Like” API failures	Concurrent likes on a hot post	Acknowledged, open
Social 4	Indexing failures	Bad data formats	Some of them fixed
MrCMS	Crash after image upload and subsequent restart of the application (Issue# 43)	Auto-generated thumbnail file name too long	Acknowledged, investigating
CMSFoundation	Failure to save edited image (Issue# 321)	Concurrent file edit and delete	Acknowledged, open
Massive	Slow request (Issue# 270)	Unoptimal use of Await	Fixed and closed
Nancy	Slow request (Issue# 2623)	Redundant Task method calls	Fixed and closed

Table 1: Summary of previously-unknown issues found by using AUDIT.

the bug once the root cause is known is simple: check file name lengths after adding the suffixes. The issue was acknowledged by the developer.

6.1.3 Case study: CMS-Foundation

CMS-Foundation[46] is a top-rated open source CMS with more than 60K installations worldwide.

Symptoms. When an admin saves after editing an image, they occasionally get a cryptic “Failed to get image properties: check that the image is not corrupt” message. The problem recurred as the admin retried the operation.

AUDIT logs. AUDIT log showed a crucial causality through two exception chains (as the application caught and rethrew exceptions) to the file being deleted while the admin was editing the image.

Root cause and fix. While the admin was editing the image, another admin deleted it, leading to a race condition. One way to fix this behavior is to use locking to prevent two admins from performing conflicting operations. The issue was acknowledged by the developers.

We now summarize two case studies demonstrating AUDIT’s value in diagnosing performance problems.

6.1.4 Case study: Massive

Massive[47] is a dynamic MicroORM and a showcasing project for ASP .NET. Massive is popular and active on GitHub, with 1.6K stars and 330 forks.

Symptoms. Slow requests for certain inputs.

AUDIT logs. AUDIT produced RAG for the slow requests, as well as input parameters and return values of 5 top-ranked methods.

Root cause and fix. The top two methods ranked by AUDIT constituted 80% of the latency for some inputs. These methods query a backend database. Input parameters (i.e., query string) of the methods indicated that the method calls are independent (we confirmed this by looking at the code), yet Massive runs them in sequence. We modified the code to call both methods in parallel. This simple change resulted in a 1.37× speedup of the query in our deployment. We filed this potential optimization on GitHub and this issue was acknowledged and fixed.

6.1.5 Case study: Nancy

Nancy[48] is “a lightweight, low-ceremony, framework for building HTTP based services on .NET Framework/-Core and Mono”. Nancy is also popular on GitHub, with 5.8K stars, 1.3K forks, and more than 250 contributors.

Symptoms. Some requests were slow.

AUDIT logs. AUDIT’s log identified RAG and top-blamed method calls for the slow requests.

Root cause and fix. The top-blamed method calls, that constitued signficant part of the latency, were expensive and redundant [42]. We therefore changed the code by simply removing the redundant code, without affecting semantics of the code. This reduced average latency of the Nancy website from 1.73ms to 1.27ms with our deployment, a 1.36× improvement. We have reported this issue to Nancy developers, who have quickly acknowledged and fixed it. This, again, shows effectiveness of AUDIT’s blame ranking algorithm.

6.2 Blame ranking algorithm

We compare AUDIT’s blame ranking algorithm with three other algorithms: (1) NPT [14] that distributes running time evenly among concurrently running methods and ranks methods based on their total time, (2) Top critical methods (TCM), which ranks methods based on their execution time on critical path, and (3) Iterative Logical Zeroing (ILZ), an extension of Logical Zeroing [16]. ILZ first selects the method that, if finished in zero time, would have the maximum reduction in end-to-end latency. It then selects the second method *after* setting the first method’s execution time to zero, and so on.

We consider four common code patterns observed in 11 different open source TAP applications and tutorials. Figure 4 shows corresponding RAGs. Two developers manually studied the applications and RAGs and identified the methods they would log to troubleshoot performance misbehaviors. Methods identified by both the developers are used as baseline.

Table 2 shows top-3 methods identified by different algorithms (and the baseline). TCM and ILZ fail to identify methods not on critical paths (e.g., Scenario 3). NPT

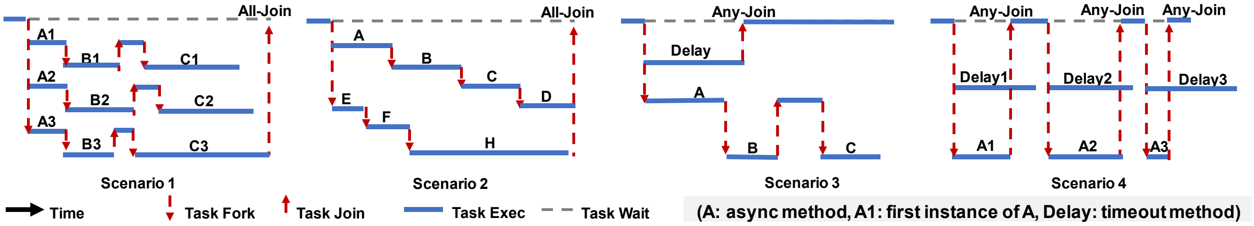


Figure 4: Common code patterns in TAP. Scenario 1 (found in Social, [49, 50]) starts parallel tasks with same code path and awaits all to finish. Scenario 2 (found in [51, 52]) starts several different tasks (which in turns fires up more children tasks) and they could finish close to each other. Scenario 3 (found in [53, 54]) starts a task and waits for a timeout. Scenario 4 (found in [20, 55, 56, 57]) retries a failed task a few times, and each trial is guarded with a timeout.

Algorithm	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Total
Baseline	C_3,B_3,A_3	H,B,A	A,C,B	A_1,A_2,A_3	–
NPT	C_3,C_2,C_1	H,B,A	$A,\mathbb{D},C$	$\mathbb{D}_3,\mathbb{D}_2,\mathbb{D}_1$	6/12
TCM	C_3,B_3,A_3	B,A,C	$\mathbb{D}$	A_2,A_1,A_3	7/12
ILZ	C_3,C_2,C_1	B,H,A	$\mathbb{D}$	A_2,A_1,A_3	7/12
AUDIT	C_3,B_3,A_3	H,B,A	$\mathbb{D},A,B$	$A_1,A_2,\mathbb{D}_3$ $\mathbb{D}_1,\mathbb{D}_2,A_3$	11/12

Table 2: Top 3 blamed methods identified by various algorithms for scenarios in Figure 4. ($\mathbb{D}$ = *Delay*.)

fail to find important methods on the critical path (e.g., Scenario 4). Last column of the table shows how many of the developer-desired methods (baseline) are identified by different algorithms. Overall, AUDIT performs better – it identified 11 out of 12 methods marked by developers; while other algorithms identified 6-7 methods only. The only scenario where AUDIT failed to identify an important method C is Scenario 3, where C does neither fall on a critical path nor overlap or interfere with any method on the critical path.

6.3 Runtime overhead

We now evaluate runtime overhead of AUDIT running on a Windows 10 D8S_V3 instance, on Microsoft Azure. Web applications are hosted using ASP.NET 5 on IIS 10.0. SQL Server 2016 is used as database.

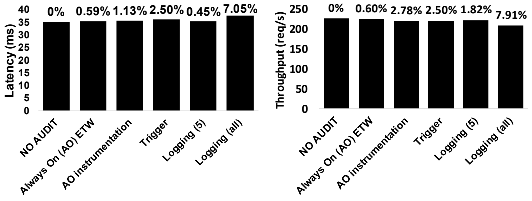
6.3.1 Micro benchmark results

Considerable design effort went in reducing the always-on overhead of AUDIT. We measure the overhead with a simple benchmark application that waits on two consecutive tasks. To measure the overhead of AUDIT's exception handling mechanism, we modified the application such that the async task throw an exception that the main task catches. Finally, to measure the *lower bound* on the cost of dynamic instrumentation, we instrumented an empty callback at the beginning and the end of each function with no parameter.

Table 3 shows AUDIT overhead numbers averaged over 100k runs. As shown, AUDIT's always-on ETW monitoring incurs small overhead – tens of μs per task.

	Without Exception	With Exception
Always-On ETW **Overhead**	15.56μs **+13.96μs/task**	112.2μs **+19.2μs/task**
Always-On INST **Overhead**	91.5μs **+89.9μs/method**	152μs **+59μs/method**
Trigger **Overhead**	29.66μs **+28.06μs/task**	283μs **+190μs/task**
Logging **Overhead**	93.5μs **+90.9μs/method**	148μs **+55μs/method**

Table 3: AUDIT overhead on benchmark application.



(a) Latency (b) Throughput

Figure 5: AUDIT overhead for Massive.

The overhead is acceptable for modern-day cloud applications that contain either compute-intensive or I/O-intensive tasks that typically run for tens of milliseconds or more. Always-on monitoring with instrumentation (Always-On INST) and metadata propagation incurs higher overhead mainly from instrumentation cost.[5] AUDIT significant lowers always-on monitoring overhead by leveraging ETW in TAP applications. The overhead is also higher immediately after a trigger fires (for constructing RAG and computing blames). This cost is acceptable as triggers are fired infrequently. Finally, logging has the highest overhead. Even an empty callback incurs hundreds of μs; serializing and logging method parameters, return values, stacktrace, etc. and writing to storage will add more overhead. This overhead clearly motivates the need for blame-proportional logging, which limits the number of logging methods and the duration of logging.

[5] Our measurement shows accessing an integer from ITS takes about 100ns and propagating an integer across thread costs 800ns, with a base cost of 700ns

6.3.2 Overheads for real applications

We measured AUDIT's overhead on Massive and Social, two TAP applications we used in our case studies. To emulate AUDIT's overhead on non-TAP applications in the always-on monitoring phase, we use AUDIT with and without its TAP optimizations (§4). We report maximum throughput and average query latency at maximum throughput over 5000 requests. We use a trigger to fire when latency is 2× the average latency (over 1 minute) and to log method parameters and return values.

Figure 5 shows the results for Massive, with a reasonably complex request that comes with Massive, involving 55 method invocations. Without TAP-optimizations, AUDIT always-on monitoring increases latency by 1.1% and reduces throughput by 2.8%. The overhead is smaller for simpler requests (with fewer methods) and is acceptable in many non-TAP applications in production. The overhead is significantly smaller with TAP-optimizations: latency and throughput are affected only by $< 0.6\%$, showing effectiveness of the optimizations.

Overhead of the trigger phase is slightly larger ($+2.5\%$ latency and -2.5% throughput). Logging all methods decreases throughput by 8% and increases latency by 7%. The high overhead is mainly due to serializing method parameters and return values of 55 dynamically invoked methods. Logging at only five top-blamed methods, however, has much smaller overhead (-0.45% latency and -1.8% throughput). This again highlights the value of logging only for a short period of time, and only a small number of top methods.

For Social, we used a complex request involving 795 method invocations. With TAP optimizations, latency and throughput overheads of always-on phase is within the measurement noise ($< 0.1\%$). Without the optimizations, the overhead of always-on is 4.3%, due to instrumentation overhead of 795 method invocations. Trigger phase incurs 4.1% overhead. Logging, again is the most expensive phase, causing 5.3% overhead.

7 Related work

In previous sections, we discussed prior work related to AUDIT's triggers (§3.1), request tracing (§4), dynamic instrumentation (§3.4), and blame ranking (§3.3). We now discuss additional related work.

AUDIT triggers are in spirit similar to datacenter network-related triggers used in Trumpet [58], but are designed for logging cloud and web applications.

Collecting effective logs and reducing logging overhead have been an important topic of research. Errlog [2] proactively adds appropriate logging statements into source code and uses adaptive sampling to reduce runtime overhead. In contrast, AUDIT dynamically instruments unmodified application binary and uses triggers rather than sampling to decide when to log. *Log*^{2} [4] enables logging within an overhead budget. Unlike AUDIT, it uses static instrumentation, continuous logging, and decides only *whether* (not *what*) to log. Several recent works investigate what should be logged for effective troubleshooting [3, 13], and AUDIT incorporates their findings in its design. Several recent proposals enhance and analyze *existing* log messages for failure diagnosis [59, 60, 61, 62], and are orthogonal to AUDIT.

Pivot Tracing [8] is closely related, but complimentary to AUDIT. It gives users, at runtime, the ability to define arbitrary metrics and aggregate them using relational operators. Unlike AUDIT, Pivot Tracing requires users to explicitly specify tracepoints to instrument and to interactively enable and disable instrumentation. Techniques from Pivot Tracing could be used to further enhance AUDIT; e.g., if implemented, happen-before join could be used as a trigger condition and baggage could be used to trace related methods across machine boundaries.

AUDIT's techniques for identifying methods related to a misbehaving request is related to end-to-end causal tracing [1, 24, 25, 26, 27, 28]. Existing solutions use instrumentation and metadata propagation; in contrast, AUDIT can also leverage cheap system events. To keep overhead acceptable in production, prior works trace coarse-grained tracepoints [1, 24], or fine-grained but a small number of carefully chosen tracepoints (which requires deep application knowledge) [26], and/or a small sample of requests [1]. In contrast, AUDIT traces all requests at method granularity, along with forks and joins of their execution.

Adaptive bug isolation [63], like AUDIT, adapts instrumentation during runtime. However, AUDIT's adaptation can be triggered by a single request (rather than statistical analysis of many requests, as in many other statistical debugging techniques [10, 64]), can work at a much finer temporal granularity (logging only for a small window of time), and has much better selectivity of logging methods due to causal tracking.

8 Conclusions

We presented AUDIT, a system for troubleshooting transiently-recurring errors in cloud-based production systems through *blame-proportional logging*, a novel mechanism with which logging information generated by a method over a period of time is proportional to how often it is blamed for various misbehaviors. AUDIT lets a developer write declarative triggers, specifying what to log and on what misbehavior, without specifying where to collect the logs. We have implemented AUDIT and evaluated it with five mature open source and commercial applications, for which AUDIT identified previously unknown issues causing slow responses and application crashes. All the issues are reported to developers, who have acknowledged or fixed them.

References

[1] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Technical report, Google, 2010.

[2] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 293–306, Hollywood, CA, 2012. USENIX.

[3] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 24–33. ACM, 2014.

[4] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 139–150, Santa Clara, CA, 2015. USENIX Association.

[5] Gerd Zellweger, Denny Lin, and Timothy Roscoe. So many performance events, so little time. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, pages 14:1–14:9, New York, NY, USA, 2016. ACM.

[6] Rui Ding, Qiang Fu, Jian-Guang Lou, Qingwei Lin, Dongmei Zhang, and Tao Xie. Mining historical issue repositories to heal large-scale online service systems. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 311–322, 2014.

[7] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):13, 2012.

[8] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393, New York, NY, USA, 2015. ACM.

[9] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[10] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 34–44. IEEE, 2009.

[11] Async stacktraceex 1.0.1.1. https://www.nuget.org/packages/AsyncStackTraceEx/, 2017.

[12] Async programming async causality chain tracking. https://msdn.microsoft.com/en-us/magazine/jj891052.aspx, 2017.

[13] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 415–425. IEEE, 2015.

[14] Thomas E. Anderson and Edward D. Lazowska. Quartz: a tool for tuning parallel program performance. In *Acm Sigmetrics Conference on Measurement & Modeling of Computer Systems*, pages 115–125, 1990.

[15] Jeffrey K. Hollingsworth and Barton P. Miller. Slack: A new performance metric for parallel programs. *University of Wisconsin-Madison Department of Computer Sciences*, 1970.

[16] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. S. Lim, and T. Torzewski. Ips-2: The second generation of a parallel program measurement system. *Parallel & Distributed Systems IEEE Transactions on*, 1(2):206–217, 1990.

[17] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 184–197, New York, NY, USA, 2015. ACM.

[18] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. *SIGARCH Comput. Archit. News*, 44(2):517–530, March 2016.

[19] Retry pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/retry, 2017.

[20] Cleanest way to write retry logic? https://stackoverflow.com/questions/1563191/cleanest-way-to-write-retry-logic.

[21] Matthew Merzbacher and Dan Patterson. Measuring end-user availability on the web: Practical experience. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 473–477. IEEE, 2002.

[22] Jennifer Widom and Stefano Ceri. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

[23] Ifttt. http://ifttt.com.

[24] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[25] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI '11, 2011.

[26] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: tracking activity in a distributed storage system. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 3–14. ACM, 2006.

[27] Raja R Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R Ganger. So, you want to trace your distributed system? key design s from years of practical experience. Technical report, Technical Report, CMU-PDL-14, 2014.

[28] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proceedings of ACM SIGCOMM 2016*, August 2016.

[29] Vara Prasad, William Cohen, FC Eigler, Martin Hunt, Jim Keniston, and J Chen. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*. Citeseer, 2005.

[30] What is callback hell? https://www.quora.com/What-is-callback-hell,, 2017.

[31] Asynchronous programming with the aws sdk for java. https://aws.amazon.com/articles/5496117154196801.

[32] Amazon web services asynchronous apis for .net. http://docs.aws.amazon.com/sdk-for-net/v3/developer-guide/sdk-net-async-api.html.

[33] Etw tracing. https://msdn.microsoft.com/en-us/library/ms751538.aspx.

[34] Nodejs v84̇0̇ documentation. https://nodejs.org/api/async_hooks.html, 2017.

[35] Asyncio decorator. https://gist.github.com/Integralist/77d73b2380e4645b564c28c53fae71fb#file-python-asyncio-timing-decorator-py-L28, 2017.

[36] Rxjava debug plugin. https://github.com/ReactiveX/RxJavaDebug.

[37] Long stacktraces. https://github.com/tlrobinson/long-stack-traces,, 2017.

[38] Appdomain.firstchanceexception event. https://msdn.microsoft.com/en-us/library/system.appdomain.firstchanceexception(v=vs.110).aspx, 2017.

[39] Thread.uncaughtexceptionhandler (java platform se 7) - oracle. http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.UncaughtExceptionHandler.html.

[40] Detect and log all java security exceptions. https://freckles.blob.core.windows.net/sites/marketing/media/assets/partners/brixbits/securityanalyzer_datasheet_2015_02.pdf.

[41] rejectionhandled event reference — mdn. https://developer.mozilla.org/en-US/docs/Web/Events/rejectionhandled.

[42] .net reference source. https://referencesource.microsoft.com/, 2017.

[43] tasklocals 0.2. https://pypi.python.org/pypi/tasklocals/.

[44] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[45] Mrcms/mrcms. https://github.com/MrCMS/MrCMS, 2017.

[46] Orckestra/cms-foundation. https://github.com/Orckestra/CMS-Foundation, 2017.

[47] Massive. https://github.com/FransBouma/Massive,, 2017.

[48] Nancy. https://github.com/NancyFx/Nancy, 2017.

[49] Codehub issueview.cs. https://github.com/CodeHubApp/CodeHub/blob/ee9b2acacab461730cf946836f5dff149908f8ad/CodeHub.iOS/Views/Issues/IssueView.cs#L276.

[50] Blazor/src/anglesharp/extensions/documentextensions.cs. https://github.com/SteveSanderson/Blazor/blob/749bae9def3ccb57006da1b155e46ea3e4c62c0f/src/AngleSharp/Extensions/DocumentExtensions.cs#L261.

[51] eshoponcontainer. https://github.com/dotnet-architecture/eShopOnContainers/search?utf8=%E2%9C%93&q=WhenAll&type=.

[52] Massive.shared.async.cs. https://github.com/FransBouma/Massive/blob/d8135f6ed44f36418ab9ee78f9cb14e023778d30/src/Massive.Shared.Async.cs#L862.

[53] Codehub whenany example. https://github.com/CodeHubApp/CodeHub/blob/e8513b052ba34ab54b7d08e08adbc4dbd3ceeac1/CodeHub.iOS/Services/InAppPurchaseService.cs#L57.

[54] Reactivewindows whenany example. https://github.com/Microsoft/react-native-windows/search?utf8=%E2%9C%93&q=whenany&type.

[55] Polly. https://github.com/App-vNext/Polly.

[56] Implementing the retry pattern for async tasks in c#. https://alastaircrabtree.com/implementing-the-retry-pattern-for-async-tasks-in-c/.

[57] Retry a task multiple times based on user input in case of an exception in task. https://stackoverflow.com/questions/10490307/retry-a-task-multiple-times-based-on-user-input-in-case-of-an

[58] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 129–143. ACM, 2016.

[59] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. *ACM SIGOPS Operating Systems Review*, 50(2):489–502, 2016.

[60] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ACM SIGARCH computer architecture news*, volume 38, pages 143–154. ACM, 2010.

[61] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30(1):4, 2012.

[62] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132. ACM, 2009.

[63] Piramanayagam Arumuga Nainar and Ben Liblit. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 255–264. ACM, 2010.

[64] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. *SIGPLAN Not.*, 49(10):561–578, October 2014.

NanoLog: A Nanosecond Scale Logging System

Stephen Yang
Stanford University

Seo Jin Park
Stanford University

John Ousterhout
Stanford University

Abstract

NanoLog is a nanosecond scale logging system that is 1-2 orders of magnitude faster than existing logging systems such as Log4j2, spdlog, Boost log or Event Tracing for Windows. The system achieves a throughput up to 80 million log messages per second for simple messages and has a typical log invocation overhead of 8 nanoseconds in microbenchmarks and 18 nanoseconds in applications, despite exposing a traditional printf-like API. NanoLog achieves this low latency and high throughput by shifting work out of the runtime hot path and into the compilation and post-execution phases of the application. More specifically, it slims down user log messages at compile-time by extracting static log components, outputs the log in a compacted, binary format at runtime, and utilizes an offline process to re-inflate the compacted logs. Additionally, log analytic applications can directly consume the compacted log and see a performance improvement of over 8x due to I/O savings. Overall, the lower cost of NanoLog allows developers to log more often, log in more detail, and use logging in low-latency production settings where traditional logging mechanisms are too expensive.

1 Introduction

Logging plays an important role in production software systems, and it is particularly important for large-scale distributed systems running in datacenters. Log messages record interesting events during the execution of a system, which serve several purposes. After a crash, logs are often the best available tool for debugging the root cause. In addition, logs can be analyzed to provide visibility into a system's behavior, including its load and performance, the effectiveness of its policies, and rates of recoverable errors. Logs also provide a valuable source of data about user behavior and preferences, which can be mined for business purposes. The more events that are recorded in a log, the more valuable it becomes.

Unfortunately, logging today is expensive. Just formatting a simple log message takes on the order of one microsecond in typical logging systems. Additionally, each log message typically occupies 50-100 bytes, so available I/O bandwidth also limits the rate at which log messages can be recorded. As a result, developers are often forced to make painful choices about which events to log; this impacts their ability to debug problems and understand system behavior.

Slow logging is such a problem today that software development organizations find themselves removing valuable log messages to maintain performance. According to our contacts at Google[7] and VMware[28], a considerable amount of time is spent in code reviews discussing whether to keep log messages or remove them for performance. Additionally, this process culls a lot of useful debugging information, resulting in many more person hours spent later debugging. Logging itself is expensive, but lacking proper logging is *very* expensive.

The problem is exacerbated by the current trend towards low-latency applications and micro-services. Systems such as Redis [34], FaRM [4], MICA[18] and RAMCloud [29] can process requests in as little as 1-2 microseconds; with today's logging systems, these systems cannot log events at the granularity of individual requests. This mismatch makes it difficult or impossible for companies to deploy low-latency services. One industry partner informed us that their company will not deploy low latency systems until there are logging systems fast enough to be used with them [7].

NanoLog is a new, open-source [47] logging system that is 1-2 orders of magnitude faster than existing systems such as Log4j2 [43], spdlog [38], glog [11], Boost Log [2], or Event Tracing for Windows [31]. NanoLog retains the convenient printf[33]-like API of existing logging systems, but it offers a throughput of around 80 million messages per second for simple log messages, with a caller latency of only 8 nanoseconds in microbenchmarks. For reference, Log4j2 only achieves a throughput of 1.5 million messages per second with latencies in the hundreds of nanoseconds for the same microbenchmark.

NanoLog achieves this performance by shifting work out of the runtime hot path and into the compilation and post-execution phases of the application:

- It rewrites logging statements at compile time to remove static information and defers expensive message formatting until the post-execution phase. This dramatically reduces the computation and I/O bandwidth requirements at runtime.
- It compiles specialized code for each log message to handle its dynamic arguments efficiently. This avoids runtime parsing of log messages and encoding argument types.
- It uses a lightweight compaction scheme and outputs the log out-of-order to save I/O and processing at runtime.
- It uses a postprocessor to combine compacted log data with extracted static information to generate

```
NANO_LOG(NOTICE, "Creating table ’%s’ with id %d", name, tableId);

2017/3/18 21:35:16.554575617 TableManager.cc:1031 NOTICE[4]: Creating table ’orders’ with id 11
```

Figure 1: A typical logging statement (top) and the resulting output in the log file (bottom). “NOTICE” is a log severity level and “[4]” is a thread identifier.

human-readable logs. In addition, aggregation and analytics can be performed directly on the compacted log, which improves throughput by over 8x.

2 Background and Motivation

Logging systems allow developers to generate a human-readable trace of an application during its execution. Most logging systems provide facilities similar to those in Figure 1. The developer annotates system code with logging statements. Each logging statement uses a printf-like interface[33] to specify a static string indicating what just happened and also some runtime data associated with the event. The logging system then adds supplemental information such as the time when the event occurred, the source code file and line number of the logging statement, a severity level, and the identifier of the logging thread.

The simplest implementation of logging is to output each log message synchronously, inline with the execution of the application. This approach has relatively low performance, for two reasons. First, formatting a log message typically takes 0.5-1 µs (1000-2000 cycles). In a low latency server, this could represent a significant fraction of the total service time for a request. Second, the I/O is expensive. Log messages are typically 50-100 bytes long, so a flash drive with 250 Mbytes/sec bandwidth can only absorb a few million messages per second. In addition, the application will occasionally have to make kernel calls to flush the log buffer, which will introduce additional delays.

The most common solution to these problems is to move the expensive operations to a separate thread. For example, I/O can be performed in a background thread: the main application thread writes log messages to a buffer in memory, and the background thread makes the kernel calls to write the buffer to a file. This allows I/O to happen in parallel with application execution. Some systems, such as TimeTrace in PerfUtils [32], also offload the formatting to the background thread by packaging all the arguments into an executable lambda, which is evaluated by the background thread to format the message.

Unfortunately, moving operations to a background thread has limited benefit because the operations must still be carried out while the application is running. If log messages are generated at a rate faster than the background thread can process them (either because of I/O or CPU limitations), then either the application must eventually block, or it must discard log messages. Neither of these options is attractive. Blocking is particularly unappealing for low-latency systems because it can result in long tail latencies or even, in some situations, the appearance that a server has crashed.

In general, developers must ensure that an application doesn’t generate log messages faster than they can be processed. One approach is to filter log messages according to their severity level; the threshold might be higher in a production environment than when testing. Another possible approach is to sample log messages at random, but this may cause key messages (such as those identifying a crash) to be lost. The final (but not uncommon) recourse is a social process whereby developers determine which log messages are most important and remove the less critical ones to improve performance. Unfortunately, all of these approaches compromise visibility to get around the limitations of the logging system.

The design of NanoLog grew out of two observations about logging. The first observation is that fully-formatted human-readable messages don’t necessarily need to be produced inside the application. Instead, the application could log the raw components of each message and the human-readable messages could be generated later, if/when a human needs them. Many logs are never read by humans, in which case the message formatting step could be skipped. When logs are read, only a small fraction of the messages are typically examined, such as those around the time of a crash, so only a small fraction of logs needs to be formatted. And finally, many logs are processed by analytics engines. In this case, it is much faster to process the raw data than a human-readable version of the log.

The second observation is that log messages are fairly redundant and most of their content is static. For example, in the log message in Figure 1, the only dynamic parts of the message are the time, the thread identifier, and the values of the `name` and `tableId` variables. All of the other information is known at compile-time and is repeated in every invocation of that logging statement. It should be possible to catalog all the static information at compile-time and output it just once for the postprocessor. The postprocessor can reincorporate the static information when it formats the human-readable messages. This approach dramatically reduces the amount of information that the application must log, thereby allowing the application to log messages at a much higher rate.

The remainder of this paper describes how NanoLog capitalizes on these observations to improve logging performance by 1-2 orders of magnitude.

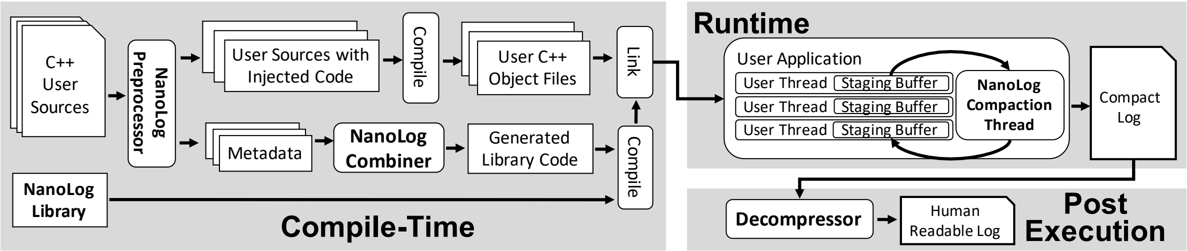


Figure 2: Overview of the NanoLog system. At compile time, the user sources are passed through the NanoLog preprocessor, which injects optimized logging code into the application and generates a metadata file for each source file. The modified user code is then compiled to produce C++ object files. The metadata files are aggregated by the NanoLog combiner to build a portion of the NanoLog Library. The NanoLog library is then compiled and linked with the user object files to create an application executable and a decompressor application. At runtime, the user application threads interact with the NanoLog staging buffers and background compaction thread to produce a compact log. At post execution, the compact log is passed into the decompressor to generate a final, human-readable log file.

3 Overview

NanoLog's low latency comes from performing work at compile-time to extract static components from log messages and deferring formating to an off-line process. As a result, the NanoLog system decomposes into three components as shown in Figure 2:

Preprocessor/Combiner: extracts and catalogs static components from log messages at compile-time, replaces original logging statements with optimized code, generates a unique compaction function for each log message, and generates a function to output the dictionary of static information.

Runtime Library: provides the infrastructure to buffer log messages from multiple logging threads and outputs the log in a compact, binary format using the generated compaction and dictionary functions.

Decompressor: recombines the compact, binary log file with the static information in the dictionary to either inflate the logs to a human-readable format or run analytics over the log contents.

Users of NanoLog interact with the system in the following fashion. First, they embed `NANO_LOG()` function calls in their C++ applications where they'd like log messages. The function has a signature similar to printf [17, 33] and supports all the features of printf with the exception of the "`%n`" specifier, which requires dynamic computation. Next, users integrate into their GNUmakefiles [40] a macro provided by NanoLog that serves as a drop-in replacement for a compiler invocation, such as g++. This macro will invoke the NanoLog preprocessor and combiner on the user's behalf and generate two executables: the user application linked against the NanoLog library, and a decompressor executable to inflate/run analytics over the compact log files. As the application runs, a compacted log is generated. Finally, the NanoLog decompressor can be invoked to read the compacted log and produce a human-readable log.

4 Detailed Design

We implemented the NanoLog system for C++ applications and this section describes the design in detail.

4.1 Preprocessor

The NanoLog preprocessor interposes in the compilation process of the user application (Figure 2). It processes the user source files and generates a *metadata file* and a modified source file for each user source file. The modified source files are then compiled into object files. Before the final link step for the application, the NanoLog combiner reads all the metadata files and generates an additional C++ source file that is compiled into the NanoLog Runtime Library. This library is then linked into the modified user application.

In order to improve the performance of logging, the NanoLog preprocessor analyzes the `NANO_LOG()` statements in the source code and replaces them with faster code. The replacement code provides three benefits. First, it reduces I/O bandwidth by logging only information that cannot be known until runtime. Second, NanoLog logs information in a compacted form. Third, the replacement code executes much more quickly than the original code. For example, it need not combine the dynamic data with the format string, or convert binary values to strings; data is logged in a binary format. The preprocessor also extracts type and order information from the format string (e.g., a `"%d %f"` format string indicates that the log function should encode an integer followed by a float). This allows the preprocessor to generate more efficient code that accepts and processes exactly the arguments provided to the log message. Type safety is ensured by leveraging the GNU `format` attribute compiler extension [10].

The NanoLog preprocessor generates two functions for each `NANO_LOG()` statement. The first function, `record()`, is invoked in lieu of the original `NANO_LOG()` statement. It records the dynamic information associated with the log message into an in-

```
inline void record(buffer, name, tableId) {
  // Unique identifier for this log statement;
  // the actual value is computed by the combiner.
  extern const int _logId_TableManager_cc_line1031;

  buffer.push<int>(_logId_TableManager_cc_line1031);
  buffer.pushTime();
  buffer.pushString(name);
  buffer.push<int>(tableId);
}

inline void compact(buffer, char *out) {
  pack<int>(buffer, out); // logId
  packTime(buffer, out); // time
  packString(buffer, out); // string name
  pack<int>(buffer, out); // tableId
}
```

Figure 3: Sample code generated by the NanoLog preprocessor and combiner for the log message in Figure 1. The `record()` function stores the dynamic log data to a buffer and `compact()` compacts the buffer's contents to an output character array.

memory buffer. The second function, `compact()`, is invoked by the NanoLog background compaction thread to compact the recorded data for more efficient I/O.

Figure 3 shows slightly simplified versions of the functions generated for the `NANO_LOG()` statement in Figure 1. The `record()` function performs the absolute minimum amount of work needed to save the log statement's dynamic data in a buffer. The invocation time is read using Intel's RDTSC instruction, which utilizes the CPU's fine grain Time Stamp Counter [30]. The only static information it records is a unique identifier for the `NANO_LOG()` statement, which is used by the NanoLog runtime background thread to invoke the appropriate `compact()` function and the decompressor to retrieve the statement's static information. The types of `name` and `tableId` were determined at compile-time by the preprocessor by analyzing the "`%s`" and "`%d`" specifiers in the format string, so `record()` can invoke type-specific methods to record them.

The purpose of the `compact()` function is to reduce the number of bytes occupied by the logged data, in order to save I/O bandwidth. The preprocessor has already determined the type of each item of data, so `compact()` simply invokes a type-specific compaction method for each value. Section 4.2.2 discusses the kinds of compaction that NanoLog performs and the trade-off between compute time and compaction efficiency.

In addition to the `record()` and `compact()` functions, the preprocessor creates a dictionary entry containing all of the static information for the log statement. This includes the file name and line number of the `NANO_LOG()` statement, the severity level and format string for the log message, the types of all the dynamic values that will be logged, and the name of a variable that will hold the unique identifier for this statement.

After generating this information, the preprocessor replaces the original `NANO_LOG()` invocation in the user source with an invocation to the `record()` function. It also stores the `compact()` function and the dictionary information in a metadata file specific to the original source file.

The NanoLog combiner executes after the preprocessor has processed all the user files (Figure 2); it reads all of the metadata files created by the preprocessor and generates additional code that will become part of the NanoLog runtime library. First, the combiner assigns unique identifier values for log statements. It generates code that defines and initializes one variable to hold the identifier for each log statement (the name of the variable was specified by the preprocessor in the metatadata file). Deferring identifier assignment to the combiner allows for a tight and contiguous packing of values while allowing multiple instances of the preprocessor to process client sources in parallel without synchronization. Second, the combiner places all of the `compact()` functions from the metadata files into a function array for the NanoLog runtime to use. Third, the combiner collects the dictionary information for all of the log statements and generates code that will run during application startup and write the dictionary into the log.

4.2 NanoLog Runtime

The NanoLog runtime is a statically linked library that runs as part of the user application and decouples the low-latency application threads executing the `record()` function from high latency operations like disk I/O. It achieves this by offering low-latency staging buffers to store the results of `record()` and a background compaction thread to compress the buffers' contents and issue disk I/O.

4.2.1 Low Latency Staging Buffers

Staging buffers store the result of `record()`, which is executed by the application logging threads, and make the data available to the background compaction thread. Staging buffers have a crucial impact on performance as they are the primary interface between the logging and background threads. Thus, they must be as low latency as possible and avoid thread interactions, which can result in lock contention and cache coherency overheads.

Locking is avoided in the staging buffers by allocating a separate staging buffer per logging thread and implementing the buffers as single producer, single consumer circular queues [24]. The allocation scheme allows multiple logging threads to store data into the staging buffers without synchronization between them and the implementation allows the logging thread and background thread to operate in parallel without locking the entire structure. This design also provides a throughput benefit as the source and drain operations on a buffer can be overlapped in time.

However, even with a lockless design, the threads' accesses to shared data can still cause cache coherency delays in the CPU. More concretely, the circular queue implementation has to maintain a head position for where the log data starts and a tail position for where the data ends. The background thread modifies the head position to consume data and the logging thread modifies the tail position to add data. However, for either thread to query how much space it can use, it needs to access both variables, resulting in expensive cache coherency traffic.

NanoLog reduces cache coherency traffic in the staging buffers by performing multiple inserts or removes for each cache miss. For example, after the logging thread reads the head pointer, which probably results in a cache coherency miss since its modified by the background thread, it saves a copy in a local variable and uses the copy until all available space has been consumed. Only then does it read the head pointer again. The compaction thread caches the tail pointer in a similar fashion, so it can process all available log messages before incurring another cache miss on the tail pointer. This mechanism is safe because there is only a single reader and a single writer for each staging buffer.

Finally, the logging and background threads store their private variables on separate cachelines to avoid false sharing [1].

4.2.2 High Throughput Background Thread

To prevent the buffers from running out of space and blocking, the background thread must consume the log messages placed in the staging staging buffer as fast as possible. It achieves this by deferring expensive log processing to the post-execution decompressor application and compacting the log messages to save I/O.

The NanoLog background thread defers log formatting and chronology sorting to the post-execution application to reduce log processing latency. For comparison, consider a traditional logging system; it outputs the log messages in a human-readable format and in chronological order. The runtime formatting incurs computation costs and bloats the log message. And maintaining chronology means the background thread must either serialize all logging or sort the log messages from concurrent logging threads at runtime. Both of these operations are expensive, so the background thread performs neither of these tasks. The NanoLog background thread simply iterates through the staging buffers in round-robin fashion and for each buffer, processes the buffer's entire contents and outputs the results to disk. The processing is also non-quiescent, meaning a logging thread can record new log messages while the background thread is processing its staging buffer's contents.

Additionally, the background thread needs to perform some sort of compression on the log messages to reduce I/O latency. However, compression only makes sense if it reduces the overall end-to-end latency. In our measurements, we found that while existing compression schemes like the LZ77 algorithm [49] used by gzip [9] were very effective at reducing file size, their computation times were too high; it was often faster to output the raw log messages than to perform any sort of compression. Thus, we developed our own lightweight compression mechanism for use in the `compact()` function.

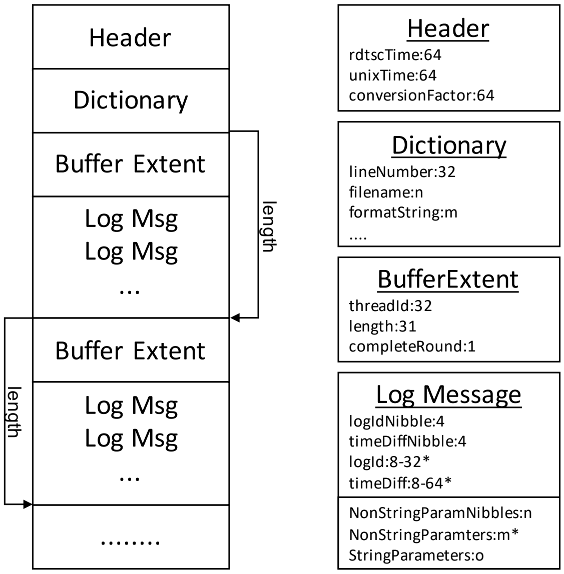


Figure 4: Layout of a compacted log file produced by the NanoLog runtime at a high level (left) and at the component level (right). As indicated by the diagram on the left, the NanoLog output file always starts with a Header and a Dictionary. The rest of the file consists of Buffer Extents. Each Buffer Extent contains log messages. On the right, the smaller text indicates field names and the digits after the colon indicate how many bits are required to represent the field. An asterisk (*) represents integer values that have been compacted and thus have a variable byte length. The lower box of "Log Message" indicates fields that are variable length (and sometimes omitted) depending on the log message's arguments.

NanoLog attempts to compact the integer types by finding the fewest number of bytes needed to represent that integer. The assumptions here are that integers are the most commonly logged type, and most integers are fairly small and do not use all the bits specified by its type. For example, a 4 byte integer of value 200 can be represented with 1 byte, so we encode it as such. To keep track of the number of bytes used for the integer, we add a *nibble* (4-bits) of metadata. Three bits of the nibble inform the algorithm how many bytes are used to encode the integer and the last bit is used to indicate a negation. The negation is useful for when small negative numbers are encoded. For example a -1 can be represented in 1 byte without ambiguity if the negation bit was set. A limitation of this scheme is that an extra half byte (nibble) is wasted in cases where the integer cannot be compacted.

Applying these techniques, the background thread produces a log file that resembles Figure 4. The first component is a header which maps the machine's Intel Time Stamp Counter [30] (TSC) value to a wall time and the conversion factor between the two. This allows the log messages to contain raw invocation TSC values and avoids wall time conversion at runtime. The header also includes the dictionary containing static information for the log messages. Following this structure are *buffer extents* which represent contiguous staging buffers that have been output at runtime and contained within them are log messages. Each buffer extent records the runtime thread id and the size of the extent (with the log messages). This allows log messages to omit the thread id and inherit it from the extent, saving bytes.

The log messages themselves are variable sized due to compaction and the number of parameters needed for the message. However, all log messages will contain at least a compacted log identifier and a compacted log invocation time relative to the last log message. This means that a simple log message with no parameters can be as small as 3 bytes (2 nibbles and 1 byte each for the log identifier and time difference). If the log message contains additional parameters, they will be encoded after the time difference in the order of all nibbles, followed by all non-string parameters (compacted and uncompacted), followed by all string parameters. The ordering of the nibbles and non-string parameters is determined by the preprocessor's generated code, but the nibbles are placed together to consolidate them. The strings are also null terminated so that we do not need to explicitly store a length for each.

4.3 Decompressor/Aggregator

The final component of the NanoLog system is the decompressor/aggregator, which takes as input the compacted log file generated by the runtime and either outputs a human-readable log file or runs aggregations over the compacted log messages. The decompressor reads the dictionary information from the log header, then it processes each of the log messages in turn. For each message, it uses the log id embedded in the file to find the corresponding dictionary entry. It then decompresses the log data as indicated in the dictionary entry and combines that data with static information from the dictionary to generate a human-readable log message. If the decompressor is being used for aggregation, it skips the message formatting step and passes the decompressed log data, along with the dictionary information, to an aggregation function.

One challenge the NanoLog decompressor has to deal with is outputting the log messages in chronological order. Recall from earlier, the NanoLog runtime outputs the log messages in staging buffer chunks called buffer extents. Each logging thread uses its own staging buffer, so log messages are ordered chronologically within an extent, but the extents for different threads can overlap in time. The decompressor must collate log entries from different extents in order to output a properly ordered log. The round-robin approach used by the compaction thread means that extents in the log are roughly ordered by time. Thus, the decompressor can process the log file sequentially. To perform the merge correctly, it must buffer two sequential extents for each logging thread at a time.

Aside from the reordering, one of the most interesting aspects of this component is the promise it holds for faster analytics. Most analytics engines have to gather the human-readable logs, parse the log messages into a binary format, and then compute on the data. Almost all the time is spent reading and parsing the log. The NanoLog aggregator speeds this up in two ways. First, the intermediate log file is extremely compact compared to its human-readable format (typically over an order of magnitude) which saves on bandwidth to read the logs. Second, the intermediate log file already stores the dynamic portions of the log in a binary format. This means that the analytics engine does not need to perform expensive string parsing. These two features mean that the aggregator component will run faster than a traditional analytics engine operating on human-readable logs.

4.4 Alternate Implementation: C++17 NanoLog

While the techniques shown in the previous section are generalizable to any programming language that exposes its source, some languages such as C++17 offer strong compile-time computation features that can be leveraged to build NanoLog without an additional preprocessor. In this section, we briefly present such an implementation for C++17. The full source for this implementation is available in our GitHub repository[47], so we will only highlight the key features here.

The primary tasks that the NanoLog preprocessor performs are (a) generating optimized functions to *record* and *compact* arguments based on types, (b) assigning unique log identifiers to each `NANO_LOG()` invocation site and (c) generating a dictionary of static log information for the postprocessor.

For the first task, we can leverage inlined variadic function templates in C++ to build optimized functions to *record* and *compact* arguments based on their types. C++11 introduced functionality to build generic functions that would specialize on the types of the arguments passed in. One variation, called "variadic templates", allows one to build functions that can accept an unbounded number of arguments and process them recursively based on type. Using these features, we can express meta `record()` and `compact()` functions which accept any number of arguments and the C++ compiler will automatically select the correct function to invoke for each argument based on type.

One problem with this mechanism is that an argument of type "`char*`" can correspond to either a "`%s`" speci-

CPU	Xeon X3470 (4x2.93 GHz cores)
RAM	24 GB DDR3 at 800 MHz
Flash	2x Samsung 850 PRO (250GB) SSDs
OS	Debian 8.3 with Linux kernel 3.16.7
OS for ETW	Windows 10 Pro 1709, Build 16299.192

Table 1: The server configuration used for benchmarking.

fier (string) or a "%p" specifier (pointer), which are handled differently. To address this issue, we leverage constant expression functions in C++17 to analyze the static format string at compile-time and build a constant expression structure that can be checked in `record()` to selectively save a pointer or string. This mechanism makes it unnecessary for NanoLog to perform the expensive format string parsing at runtime and reduces the runtime cost to a single if-check.

The second task is assignment of unique identifiers. C++17 NanoLog must discover all the `NANO_LOG()` invocation sites dynamically and associate a unique identifier with each. To do this, we leverage scoped static variables in C++; `NANO_LOG()` is defined as a macro that expands to a new scope with a static identifier variable initialized to indicate that no identifier has been assigned yet. This variable is passed by reference to the `record()` function, which checks its value and assigns a unique identifier during the first call. Future calls for this invocation site pay only for an if-check to confirm that the identifier has been assigned. The scoping of the identifier keeps it private to the invocation site and the static keyword ensures that the value persists across all invocations for the lifetime of the application.

The third task is to generate the dictionary required by the postprocessor and write it to the log. The dictionary cannot be included in the log header, since the NanoLog runtime has no knowledge of a log statement until it executes for the first time. Thus, C++17 NanoLog outputs dictionary information to the log in an incremental fashion. Whenever the runtime assigns a new unique identifier, it also collects the dictionary information for that log statement. This information is passed to the compaction thread and output in the header of the next Buffer Extent that contains the first instance of this log message. This scheme ensures that the decompressor encounters the dictionary information for a log statement before it encounters any data records for that log statement.

The benefit of this C++17 implementation is that it is easier to deploy (users no longer have to integrate the NanoLog preprocessor into their build chain), but the downsides are that it is language specific and performs slightly more work at runtime.

5 Evaluation

We implemented the NanoLog system for C++ applications. The NanoLog preprocessor and combiner comprise of 1319 lines of Python code and the NanoLog runtime library consists of 3657 lines of C++ code.

System Name	Static Chars	Integers	Floats	Strings	Others	Logs
Memcached	56.04	0.49	0.00	0.23	0.04	378
httpd	49.38	0.29	0.01	0.75	0.03	3711
linux	35.52	0.98	0.00	0.57	0.10	135119
Spark	43.32	n/a	n/a	n/a	n/a	2717
RAMCloud	46.65	1.08	0.07	0.47	0.02	1167

Table 2: Shows the average number of static characters (Static Chars) and dynamic variables in formatted log statements for five open source systems. These numbers were obtained by applying a set of heuristics to identify log statements in the source files and analyzing the embedded format strings; the numbers do not necessarily reflect runtime usage and may not include every log invocation. The "Logs" column counts the total number of log messages found. The dynamic counts are omitted for Spark since their logging system does not use format specifiers, and thus argument types could not be easily extracted. The static characters column omits format specifiers and variable references (i.e. $variables in Spark), and represents the number of characters that would be trivially saved by using NanoLog.

We evaluated the NanoLog system to answer the following questions:

- How do NanoLog's log throughput and latency compare to other modern logging systems?
- What is the throughput of the decompressor?
- How efficient is it to query the compacted log file?
- How does NanoLog perform in a real system?
- What are NanoLog's throughput bottlenecks?
- How does NanoLog's compaction scheme compare to other compression algorithms?

All experiments were conducted on quad-core machines with SATA SSDs that had a measured throughput of about 250MB/s for large writes (Table 1).

5.1 System Comparison

To compare the performance of NanoLog with other systems, we ran microbenchmarks with six log messages (shown in Table 3) selected from an open-source datacenter storage system [29].

5.1.1 Test Setup

We chose to benchmark NanoLog against Log4j2 [43], spdlog [38], glog [11], Boost log [2], and Event Tracing for Windows (ETW) [31]. We chose Log4j2 for its popularity in industry; we configured it for low latency and high throughput by using asynchronous loggers and appenders and including the LMAX Disruptor library [20]. We chose spdlog because it was the first result in an Internet search for "Fast C++ Logger"; we configured spdlog with a buffer size of 8192 entries (or 832KB). We chose glog because it is used by Google and configured it to buffer up to 30 seconds of logs. We chose Boost logging because of the popularity of Boost libraries in the C++ community; we configured Boost to use asynchronous sinks. We chose ETW because of its similarity to NanoLog; when used with Windows Software

ID	Example Output
staticString	Starting backup replica garbage collector thread
stringConcat	Opened session with coordinator at basic+udp:host=192.168.1.140,port=12246
singleInteger	Backup storage speeds (min): 181 MB/s read
twoIntegers	buffer has consumed 1032024 bytes of extra storage, current allocation: 1016544 bytes
singleDouble	Using tombstone ratio balancer with ratio = 0.4
complexFormat	Initialized InfUdDriver buffers: 50000 receive buffers (97 MB), 50 transmit buffers (0 MB), took 26.2 ms

Table 3: Log messages used to generate Figure 5 and Table 4. The underlines indicate dynamic data generated at runtime. *staticString* is a completely static log message, *stringConcat* contains a large dynamic string, and other messages are a combination of integer and floating point types. Additionally, the logging systems were configured to output each message with the context "YY-MM-DD HH:MM:SS.ns Benchmark.cc:20 DEBUG[0]:" prepended to it.

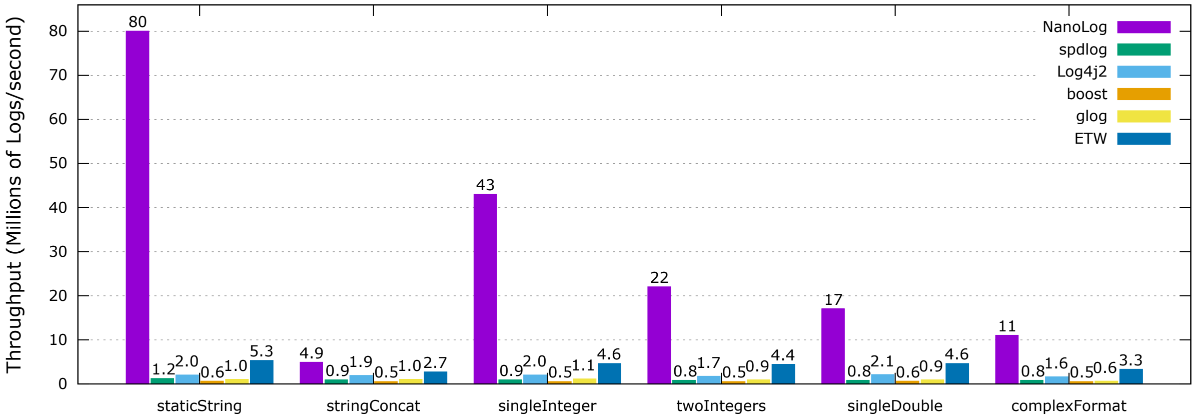


Figure 5: Shows the maximum throughput attained by various logging systems when logging a single message repeatedly. Log4j2, Boost, spdlog, and Google glog logged the message 1 million times; ETW and NanoLog logged the message 8 and 100 million times repectively to generate a log file of comparable size. The number of logging threads varied between 1-16 and the maximum throughput achieved is reported. All systems except Log4j2 include the time to flush the messages to disk in its throughput calculations (Log4j2 did not provide an API to flush the log without shutting down the logging service). The message labels on the x-axis are explained in Table 3.

Trace PreProcessor [23], the log statements are rewritten to record only variable binary data at runtime. We configured ETW with the default buffer size of 64 KB; increasing it to 1 MB did not improve its steady-state performance.

We configured each system to output similar metadata information with each log message; they prepend a date/-time, code location, log level, and thread id to each log message as shown in Figure 1. However, there are implementation differences in each system. In the time field, NanoLog and spdlog computed the fractional seconds with 9 digits of precision (nanoseconds) vs 6 for Boost-/glog and 3 for Log4j2 and ETW. In addition, Log4j2's code location information (ex. "Benchmark.cc:20") was manually encoded due to inefficiencies in its code location mechanism [45]. The other systems use the GNU C++ preprocessor macros "__LINE__" and "__FILE__" to encode the code location information.

To ensure the log messages we chose were representative of real world usage, we statically analyzed log statements from five open source systems[22, 42, 19, 44, 29]. Table 2 shows that log messages have around 45 characters of static content on average and that integers are the most common dynamic type. Strings are the second most common type, but upon closer inspection, most strings used could benefit from NanoLog's static extraction methods. They contain pretty print error messages, enumerations, object variables, and other static/formatted types. This static information could in theory be also extracted by NanoLog and replaced with an identifier. However, we leave this additional extraction of static content this to future work.

5.1.2 Throughput

Figure 5 shows the maximum throughput achieved by NanoLog, spdlog [38], Log4j2 [43], Boost [2], Google glog [11], and ETW [31]. NanoLog is faster than the other systems by 1.8x-133x. The largest performance gap between NanoLog and the other systems occurs with *staticString* and the smallest occurs with *stringConcat*.

NanoLog performs best when there is little dynamic information in the log message. This is reflected by *staticString*, a static message, in the throughput benchmark. Here, NanoLog only needs to output about 3-4 bytes per log message due to its compaction and static extraction techniques. Other systems require over an order of magnitude more bytes to represent the messages (41-90 bytes). Even ETW, which uses a preprocessor to strip messages, requires at least 41 bytes in the static string

ID	NanoLog				spdlog				Log4j2				glog				Boost				ETW			
Percentiles	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*
staticString	8	9	29	33	230	236	323	473	192	311	470	1868	1201	1229	3451	5231	1619	2338	3138	4413	180	187	242	726
stringConcat	8	9	29	33	436	494	1579	1641	230	1711	3110	6171	1235	1272	3469	5728	1833	2621	3387	5547	208	218	282	2954
singleInteger	8	9	29	35	353	358	408	824	223	321	458	1869	1250	1268	3543	5458	1963	2775	3396	7040	189	195	237	720
twoIntegers	7	8	29	44	674	698	807	1335	160	297	550	1992	1369	1420	3554	5737	2255	3167	3932	7775	200	207	237	761
singleDouble	8	9	29	34	607	637	685	1548	157	252	358	1494	2077	2135	4329	6995	2830	3479	3885	7176	187	193	248	720
complexFormat	8	8	28	33	1234	1261	1425	3360	146	233	346	1500	2570	2722	5167	8589	4175	4621	5189	9637	242	252	304	1070

Table 4: Unloaded tail latencies of NanoLog and other popular logging frameworks, measured by logging 100,000 log messages from Table 3 with a 600 nanosecond delay between log invocations to ensure that I/O is not a bottleneck. Each datum represents the 50th/90th/99th/99.9th percentile latencies measured in nanoseconds.

case. NanoLog excels with static messages, reaching a throughput of 80 million log messages per second.

NanoLog performs the worst when there's a large amount of dynamic information. This is reflected in *stringConcat*, which logs a large 39 byte dynamic string. NanoLog performs no compaction on string arguments and thus must log the entire string. This results in an output of 41-42 bytes per log message and drops throughput to about 4.9 million log messages per second.

Overall, NanoLog is faster than all other logging systems tested. This is primarily due to NanoLog consistently outputting fewer bytes per message and secondarily because NanoLog defers the formatting and sorting of log messages.

5.1.3 Latency

NanoLog lowers the logging thread's invocation latency by deferring the formatting of log messages. This effect can be seen in Table 4. NanoLog's invocation latency is 18-500x lower than other systems. In fact, NanoLog's 50/90/99th percentile latencies are all within tens of nanoseconds while the median latencies for the other systems *start* at hundreds of nanoseconds.

All of the other systems except ETW require the logging thread to either fully or partially materialize the human-readable log message before transferring control to the background thread, resulting in higher invocation latencies. NanoLog on the other hand, performs no formatting and simply pushes all arguments to the staging buffer. This means less computation and fewer bytes copied, resulting in a lower invocation latency.

Although ETW employs techniques similar to NanoLog, its latencies are much higher than those of NanoLog. We are unsure why ETW is slower than NanoLog, but one hint is that the even with the preprocessor, ETW log messages are larger than NanoLog (41 vs. 4 bytes for staticString). ETW emits extra log information such as process ids and does not use the efficient compaction mechanism of NanoLog to reduce its output.

Overall, NanoLog's unloaded invocation latency is extremely low.

5.2 Decompression

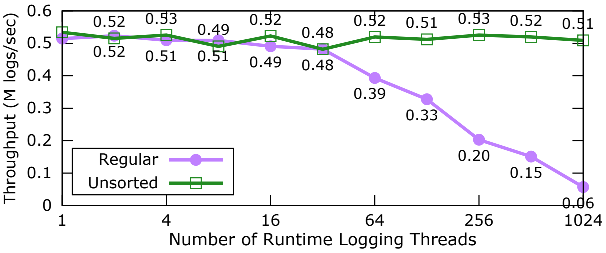


Figure 6: Impact on NanoLog's decompressor performance as the number of runtime logging threads increases. We decompressed a log file containing 2^{24} log messages (about 16M) in the format of "2017-04-06 02:03:25.000472519 Benchmark.cc:65 NOTICE[0]: Simple log message with 0 parameters". The compacted log file was 49MB and the resulting decompressed log output was 1.5GB. In the "Unsorted" measurements, the decompressor did not collate the log entries from different threads into a single chronological order.

Since the NanoLog runtime outputs the log in a binary format, it is also important to understand the performance implications of transforming it back into a human readable log format.

The decompressor currently uses a simple single-threaded implementation, which can decompress at a peak of about 0.5M log messages/sec (Figure 6). Traditional systems such as Log4j2 can achieve a higher throughput of over 2M log messages/second at runtime since they utilize all their logging threads for formatting. NanoLog's decompressor can be modified to use multiple threads to achieve higher throughput.

The throughput of the decompressor can drop if there were many runtime logging threads in the application. The reason is that the log is divided into different extents for each logging thread, and the decompressor must collate the log messages from multiple extents into a single chronological order. Figure 6 shows that decompressor can handle up to about 32 logging threads with no impact on its throughput, but throughput drops with more than 32 logging threads. This is because the decompressor uses a simple collation algorithm that compares the times for the next message from each active buffer extent (one per logging thread) in order to pick the next message to print; thus the cost per message increases linearly with the number of logging threads. Performance could be improved by using a heap for collation.

Collation is only needed if order matters during decompression. For some applications, such as analytics,

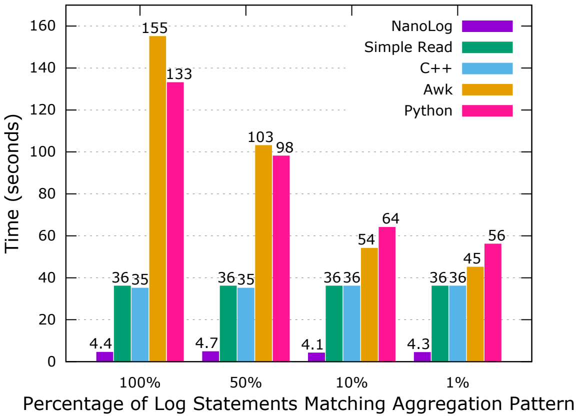


Figure 7: Execution time for a min/mean/max aggregation using various systems over 100 million log messages with a percentage of the log messages matching the target aggregation pattern "`Hello World # %d`" and the rest "`UnrelatedLog #%d`". The NanoLog system operated on a compacted file (∼747MB) and the remaining systems operated on the full, uncompressed log (∼7.6GB). The C++ application searched for the "`Hello World #`" prefix and utilized atoi() on the next word to parse the integer. The Awk and Python applications used a simple regular expression matching the prefix: "`.*Hello World # (\d+)`". "Simple Read" reads the entire log file and discards the contents. The file system cache was flushed before each run.

the order in which log messages are processed is unimportant. In these cases, collation can be skipped; Figure 6 shows that decompression throughput in this case is unaffected by the number of logging threads.

5.3 Aggregation Performance

NanoLog's compact, binary log output promises more efficient log aggregation/analytics than its full, uncompressed counterpart. To demonstrate this, we implemented a simple min/mean/max aggregation in four systems, NanoLog, C++, Awk, and Python. Conceptually, they all perform the same task; they search for the target log message "`Hello World #%d`" and perform a min/mean/max aggregation over the "`%d`" integer argument. The difference is that the latter three systems operate on the full, uncompressed version of the log while the NanoLog aggregator operates directly on the output from the NanoLog runtime.

Figure 7 shows the execution time for this aggregation over 100M log messages. NanoLog is nearly an order of magnitude faster than the other systems, taking on average 4.4 seconds to aggregate the compact log file vs. 35+ seconds for the other systems. The primary reason for NanoLog's low execution time is disk bandwidth. The compact log file only amounted to about 747MB vs. 7.6GB for the uncompressed log file. In other words, the aggregation was disk bandwidth limited and NanoLog used the least amount of disk IO. We verified this assumption with a simple C++ application that performs no aggregation and simply reads the file ("Simple Read" in the figure); its execution time lines up with the "C++" aggregator at around 36 seconds.

		No Logs	NanoLog	spdlog	RAMCloud
Throughput (kop/s)	Read	994 (100%)	809 (81%)	122 (12%)	67 (7%)
	Write	140 (100%)	137 (98%)	59 (42%)	32 (23%)
Read Latency (μs)	50%	5.19 (1.00x)	5.33 (1.03x)	8.21 (1.58x)	15.55 (3.00x)
	90%	5.56 (1.00x)	5.53 (0.99x)	8.71 (1.57x)	16.66 (3.00x)
	99%	6.15 (1.00x)	6.15 (1.00x)	9.60 (1.56x)	17.82 (2.90x)
Write Latency (μs)	50%	15.85 (1.00x)	16.33 (1.03x)	24.88 (1.57x)	45.53 (2.87x)
	90%	16.50 (1.00x)	17.08 (1.04x)	26.42 (1.60x)	47.50 (2.88x)
	99%	22.87 (1.00x)	23.74 (1.04x)	33.05 (1.45x)	59.17 (2.59x)

Table 5: Shows the impact on RAMCloud [29] performance when more intensive instrumentation is enabled. The instrumentation adds about 11-33 log statements per read/write request with 1-3 integer log arguments each. "No Logs" represents the baseline with no logging enabled. "RAMCloud" uses the internal logger while "NanoLog" and "spdlog" supplant the internal logger with their own. The percentages next to Read-/Write Latency represent percentiles and all results were measured with RAMCloud's internal benchmarks with 16 clients used in the throughput measurements. Throughput benchmarks were run for 10 seconds and latency benchmarks measured 2M operations. Each configurations was run 15 times and the best case is presented.

We also varied how often the target log message "`Hello World #%d`" occurred in the log file to see if it affects aggregation time. The compiled systems (NanoLog and C++) have a near constant cost for aggregating the log file while the interpreted systems (Awk and Python) have processing costs correlated to how often the target message occurred. More specifically, the more frequent the target message, the longer the execution time for Awk and Python. We suspect the reason is because the regular expression systems used by Awk and Python can quickly disqualify non-matching strings, but perform more expensive parsing when a match occurs. However, we did not investigate further.

Overall, the compactness of the NanoLog binary log file allows for fast aggregation.

5.4 Integration Benchmark

We integrated NanoLog and spdlog into a well instrumented open-source key value store, RAMCloud[29], and evaluated the logging systems' impact on performance using existing RAMCloud benchmarks. In keeping with the goal of increasing visibility, we enabled verbose logging and changed existing performance sampling statements in RAMCloud (normally compiled out) to always-on log statements. This added an additional 11-33 log statements per read/write request in the system. With this heavily instrumented system, we could answer the following questions: (1) how much of an improvement does NanoLog provide over other state-of-the-art systems in this scenario, (2) how does NanoLog perform in a real system compared to microbenchmarks and (3) how much does NanoLog slowdown compilation and increase binary size?

Table 5 shows that, with NanoLog, the additional instrumentation introduces only a small performance penalty. Median read-write latencies increased only by about 3-4% relative to an uninstrumented system and write throughput decreased by 2%. Read throughput sees a larger degradation (about 19%); we believe this is because read throughput is bottlenecked by RAMCloud's dispatch thread [29], which performs most of the logging. In contrast, the other logging systems incur such a high performance penalty that this level of instrumentation would probably be impractical in production: latencies increase by 1.6-3x, write throughput drops by more than half, and read throughput is reduced to roughly a tenth of the uninstrumented system (8-14x). These results show that NanoLog supports a higher level of instrumentation than other logging systems.

Using this benchmark, we can also estimate NanoLog's invocation latency when integrated in a low-latency system. For RAMCloud's read operation, the critical path emits 8 log messages out of the 11 enabled. On average, each log message increased latency by (5.33-5.19)/8 = 17.5ns. For RAMCloud's write operation, the critical path emits 27 log messages, suggesting an average latency cost of 17.7ns. These numbers are higher than the median latency of 7-8ns reported by the microbenchmarks, but they are still reasonably fast.

Lastly, we compared the compilation time and binary size of RAMCloud with and without NanoLog. Without NanoLog, building RAMCloud takes 6 minutes and results in a binary with the size of 123 MB. With NanoLog, the build time increased by 25 seconds (+7%), and the size of the binary increased to 130 MB (+6%). The dictionary of static log information amounted to 229KB for 922 log statements (∼ 248B/message). The log message count differs from Table 2 because RAMCloud compiles out log messages depending on build parameters.

5.5 Throughput Bottlenecks

NanoLog's performance is limited by I/O bandwidth in two ways. First, the I/O bandwidth itself is a bottleneck. Second, the compaction that NanoLog performs in order to reduce the I/O cost can make NanoLog compute bound as I/O speeds improve. Figure 8 explores the limits of the system by removing these bottlenecks.

Compaction plays a large role in improving NanoLog's throughput, even for our relatively fast flash devices (250MB/s). The "Full System" as described in the paper achieves a throughput of nearly 77 million operations per second while the "No Compact" system only achieves about 13 million operations per second. This is due to the 5x difference in I/O size; the full system outputs 3-4 bytes per message while the no compaction system outputs about 16 bytes per message.

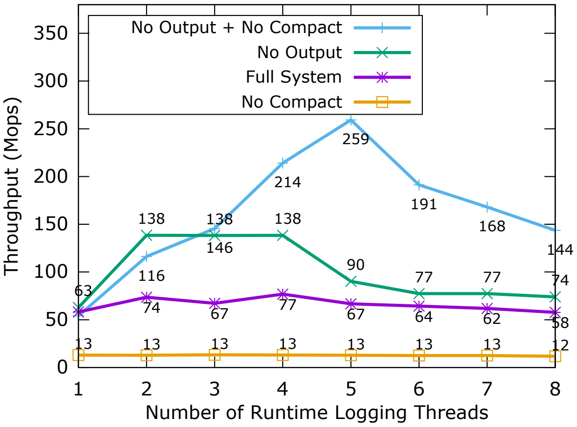


Figure 8: Runtime log message throughput acheived by the NanoLog system as the number of logging threads is increased. For each point, 2^{27} (about 134M) static messages were logged. The *Full System* is the NanoLog system as described in this paper, *No Output* pipes the log output to /dev/null, *No Compact* omits compaction in the NanoLog compression thread and directly outputs the staging buffers' contents, and *No Output + No Compact* is a combination of the the last two.

If we remove the I/O bottleneck altogether by redirecting the log file to /dev/null, NanoLog "No Output" achieves an even higher peak throughput of 138 million logs per second. At this point, the compaction becomes the bottleneck of the system. Removing both compaction and I/O allows the "No Output + No Compact" system to push upwards of 259 million operations per second.

Since the "Full System" throughput was achieved with a 250MB/s disk and the "No Output" has roughly twice the throughput, one might assume that compaction would become the bottleneck with I/O devices twice as fast as ours (500MB/s). However, that would be incorrect. To maintain the 138 million logs per second without compaction, one would need an I/O device capable of 2.24GB/s (138e6 logs/sec x 16B).

Lastly, we suspect we were unable to measure the maximum processing potential of the NanoLog compaction thread in "No Output + No Compact." Our machines only had 4 physical cores with 2 hyperthreads each; beyond 4-5, the logging threads start competing with the background thread for physical CPU resources, lowering throughput.

5.6 Compression Efficiency

NanoLog's compression mechanism is not very sophisticated in comparison to alternatives such as gzip [9] and Google snappy [37]. However, in this section we show that for logging applications, NanoLog's approach provides a better overall balance between compression efficiency and execution time.

Figure 9 compares NanoLog, gzip, and snappy using 93 test cases with varying argument types and lengths chosen to cover a range of log messages and show the

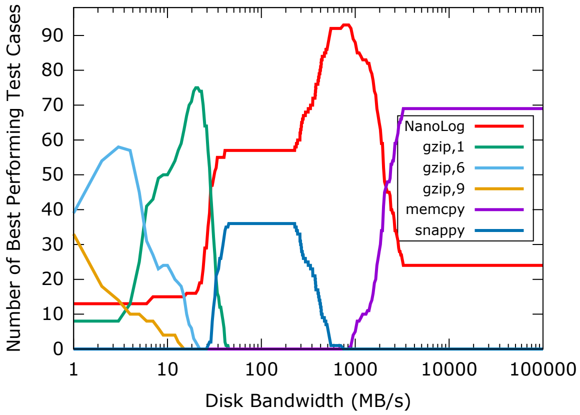


Figure 9: Shows the number of test cases (out of 93) for which a compression algorithm attained the highest throughput. Here, throughput is defined as the minimum of an algorithm's compression throughput and I/O throughput (determined by output size and bandwidth). The numbers after the "gzip" labels indicate compression level and "memcpy" represents "no compression". The input test cases were 64MB chunks of binary NanoLog logs with arguments varied in 4 dimensions: argument type (int/long/double/string), number of arguments, entropy, and value range. Strings had [10, 15, 20, 30, 45, 60, 100] characters and an entropy of "random", "zipfian" (θ=0.99), and "Top1000" (sentences generated using the top 1000 words from [26]). The numeric types had [1,2,3,4,6,10] arguments, an entropy of "random" or "sequential," and value ranges of "up to 2 bytes" and "at least half the container".

best and worst of each algorithm. For each test case and compression algorithm combination, we measured the total logging throughput at a given I/O bandwidth. Here, the throughput is determined by the lower of the compression throughput and I/O throughput (i.e. time to output the compressed data). Since the background thread overlaps the two operations, the slower operation is ultimately the bottleneck. We then counted the number of test cases where an algorithm produced highest throughput of all algorithms at a given I/O bandwidth and graphed the results in Figure 9.

From Figure 9 we see that aggressive compression only makes sense in low bandwidth situations; gzip,9 produces the best compression, but it uses so much CPU time that it only makes sense for very low bandwidth I/O devices. As I/O bandwidth increases, gzip's CPU time quickly becomes the bottleneck for throughput, and compression algorithms that don't compress as much but operate more quickly become more attractive.

NanoLog provides the highest logging throughput for most test cases in the bandwidth range for modern disks and flash drives (30–2200 MB/s). The cases where NanoLog is not the best are those involving strings and doubles, which NanoLog does not compact; snappy is better for these cases. Surprisingly, NanoLog is sometimes better than memcpy even for devices with extremely high I/O throughput. We suspect this is due to out-of-order execution[16], which can occasionally overlap NanoLog's compression with load/stores of the arguments; this makes NanoLog's compaction effectively free. Overall, NanoLog's compaction scheme is the most efficient given the capability of current I/O devices.

6 Related Work

Many frameworks and libraries have been created to increase visibility in software systems.

The system most similar to NanoLog is Event Tracing for Windows (ETW) [31] with the Windows Software Trace PreProcessor (WPP) [23], which was designed for logging inside the Windows kernel. This system was unbeknownst to us when we designed NanoLog, but WPP appears to use compilation techniques similar to NanoLog. Both use a preprocessor to rewrite log statements to record only binary data at runtime and both utilize a postprocessor to interpret logs. However, ETW with WPP does not appear to be as performant as NanoLog; in fact, it's on par with traditional logging systems with median latencies at 180ns and a throughput of 5.3Mop/s for static strings. Additionally, its postprocessor can only process messages at a rate of 10k/second while NanoLog performs at a rate of 500k/second.

There are five main differences between ETW with WPP and NanoLog: (1) ETW is a non-guaranteed logger (meaning it can drop log messages) whereas NanoLog is guaranteed. (2) ETW logs to kernel buffers and uses a separate kernel process to persist them vs. NanoLog's in-application solution. (3) The ETW postprocessor interprets a separate trace message format file to parse the logs whereas NanoLog uses dictionary information embedded in the log. (4) WPP appears to be targeted at Windows Driver Development (only available in WDK), whereas NanoLog is targeted at applications. Finally, (5) NanoLog is an open-source library [47] with public techniques that can ported to other platforms and languages while ETW is proprietary and locked to Windows only. There may be other differences (such as the use of compression) that we cannot ascertain from the documentation since ETW is closed source.

There are also general purpose, application-level loggers such as Log4j2 [43], spdlog [38], glog [11], and Boost log [2]. Like NanoLog, these systems enable applications to specify arbitrarily formatted log statements in code and provide the mechanism to persist the statements to disk. However these systems are slower than NanoLog; they materialize the human-readable log at runtime instead of deferring to post-execution (resulting in a larger log) and do not employ static analysis to generate low-latency, log specific code.

There are also implementations that attempt to provide ultra low-latency logging by restricting the data types or the number of arguments that can be logged [24, 32].

This technique reduces the amount of compute that must occur at runtime, lowering latency. However, NanoLog is able to reach the same level of performance without sacrificing flexibility by employing code generation.

Moving beyond a single machine, there are also distributed tracing tools such as Google Dapper [35], Twitter Zipkin [48], X-Trace [8], and Apache's HTrace [41]. These systems handle the additional complexity of tracking requests as they propagate across software boundaries, such as between machines or processes. In essence, these systems track causality by attaching unique request identifiers with log messages. However, these systems do not accelerate the actual runtime logging mechanism.

Once the logs are created, there are systems and machine learning services that aggregate them to provide analytics and insights [39, 3, 25, 27]. However, for compatibility, these systems typically aggregate full, human-readable logs to perform analytics. The NanoLog aggregator may be able to improve their performance by operating directly on compacted, binary logs, which saves I/O and processing time.

There are also systems that employ dynamic instrumentation [15] to gain visibility into applications at runtime such as Dtrace [14], Pivot Tracing [21], Fay [6], and Enhanced Berkley Packet Filters [13]. These systems eschew the practice of embedding static log statement at compile-time and allow for dynamic modification of the code. They allow for post-compilation insertion of instrumentation and faster iterative debugging, but the downside is that instrumentation must already be in place to enable post mortem debugging.

Lastly, it's worth mentioning that the techniques used by NanoLog and ETW are extremely similar to low-latency RPC/serialization libraries such as Thrift [36], gRPC [12], and Google Protocol Buffers [46]. These systems use a static message specification to name symbolic variables and types (not unlike NanoLog's printf format string) and generate application code to encode/decode the data into succinct I/O optimized formats (similiar to how NanoLog generates the record and compact functions). In summary, the goals and techniques used by NanoLog and RPC systems are similar in flavor, but are applied to different mediums (disk vs. network).

7 Limitations

One limitation of NanoLog is that it currently can only operate on static printf-like format strings. This means that dynamic format strings, C++ streams, and toString() methods would not benefit from NanoLog. While we don't have a performant solution for dynamic format strings, we believe that a stronger preprocessor/compiler extension may be able to extract patterns from C++ streams by looking at types and/or provide a snprintf-like function for toString() methods to generate a intermediate representation for NanoLog.

Additionally, while NanoLog is implemented in C++, we believe it can be extended to any language that exposes source code, since preprocessing and code replacement can be performed in almost any language. The only true limitation is that we would be unable to optimize any logs that are dynamically generated and evaluated (such as with JavaScript's `eval()` [5]).

NanoLog's preprocessor-based approach also creates some deployment issues, since it requires the preprocessor to be integrated in the development tool chain. C++17 NanoLog eliminates this issue using compile-time computation facilities, but not all languages can support this approach.

Lastly, NanoLog currently assumes that logs are stored in a local filesystem. However, it could easily be modified to store logs remotely (either to remotely replicated files or to a remote database). In this case, the throughput of NanoLog will be limited by the throughput of the network and/or remote storage mechanism. Most structured storage systems, such as databases or even main-memory stores, are slow enough that they would severely limit NanoLog performance.

8 Conclusion

NanoLog outperforms traditional logging systems by 1-2 orders of magnitude, both in terms of throughput and latency. It achieves this high performance by statically generating and injecting optimized, log-specific logic into the user application and deferring traditional runtime work, such as formatting and sorting of log messages, to an off-line process. This results in an optimized runtime that only needs to output a compact, binary log file, saving I/O and compute. Furthermore, this log file can be directly consumed by aggregation and log analytics applications, resulting in over an order of magnitude performance improvement due to I/O savings.

With traditional logging systems, developers often have to choose between application visibility or application performance. With the lower overhead of NanoLog, we hope developers will be able to log more often and log in more detail, making the next generation of applications more understandable.

9 Acknowledgements

We would like to thank our shepherd, Patrick Stuedi, and our anonymous reviewers for helping us improve this paper. Thanks to Collin Lee and Henry Qin for providing feedback on the design of NanoLog and Bob Felderman for refusing to use our low latency systems until we had developed better instrumentation. Lastly, thank you to the Industrial Affiliates of the Stanford Platform Lab for supporting this work. Seo Jin Park was additionally supported by Samsung Scholarship.

References

[1] Bolosky, W. J., and Scott, M. L. False Sharing and Its Effect on Shared Memory Performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4* (Berkeley, CA, USA, 1993), Sedms'93, USENIX Association, pp. 3–3.

[2] boost C++ libraries. `http://www.boost.org`.

[3] Datadog. `https://www.datadoghq.com`.

[4] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414.

[5] ECMAScript, ECMA and European Computer Manufacturers Association and others. Ecmascript language specification, 2011.

[6] Erlingsson, Ú., Peinado, M., Peter, S., Budiu, M., and Mainar-Ruiz, G. Fay: extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS) 30*, 4 (2012), 13.

[7] Felderman, B. Personal communication, June 2015. Google.

[8] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation* (2007), USENIX Association, pp. 20–20.

[9] Gailly, J.-l., and Adler, M. gzip. `http://www.gzip.org`.

[10] GNU Community. Using the GNU Compiler Collection: Declaring Attributes of Functions. `https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Function-Attributes.html`, 2002.

[11] Google. glog: Google Logging Module. `https://github.com/google/glog`.

[12] Google. gRPC: A high performance, open-source universal RPC framework. `http://www.grpc.io`.

[13] Gregg, B. Linux bpf superpowers. `http://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html`, 2016.

[14] Gregg, B., and Mauro, J. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall Professional, 2011.

[15] Hollingsworth, J. K., Miller, B. P., and Cargille, J. Dynamic program instrumentation for scalable performance tools. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the* (1994), IEEE, pp. 841–850.

[16] Intel, R. Intel 64 and ia-32 architectures optimization reference manual. *Intel Corporation, June* (2016).

[17] Jtc, I. SC22/WG14. ISO/IEC 9899: 2011. *Information Technology Programming languages C.* (2011).

[18] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. Mica: A holistic approach to fast in-memory key-value storage. USENIX.

[19] The Linux Kernel Organization. `https://www.kernel.org/nonprofit.html`, May 2018.

[20] LMAX Disruptor: High Performance Inter-Thread Messaging Library. `http://lmax-exchange.github.io/disruptor/`.

[21] Mace, J., Roelke, R., and Fonseca, R. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 378–393.

[22] memcached: a Distributed Memory Object Caching System. `http://www.memcached.org/`, Jan. 2011.

[23] Microsoft. Wpp software tracing. `https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/wpp-software-tracing`, 2007.

[24] Mortoray, E. Wait-free queueing and ultra-low latency logging. `https://mortoray.com/2014/05/29/wait-free-queueing-and-ultra-low-latency-logging/`, 2014.

[25] Nagaraj, K., Killian, C., and Neville, J. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 26–26.

[26] NORVIG, P. *Natural Language Corpus Data: Beautiful Data*, 2011 (accessed January 3, 2018).

[27] OLINER, A., GANAPATHI, A., AND XU, W. Advances and challenges in log analysis. *Communications of the ACM 55*, 2 (2012), 55–61.

[28] OTT, D. Personal communication, June 2015. VMWare.

[29] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., ET AL. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 7.

[30] PAOLONI, G. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. *Intel Corporation, September 123* (2010).

[31] PARK, I., AND BUCH, R. Improve Debugging And Performance Tuning With ETW. *MSDN Magazine*, April 2007 (2007).

[32] Performance Utilities. `https://github.com/PlatformLab/PerfUtils`.

[33] printf - C++ Reference. `http://www.cplusplus.com/reference/cstdio/printf/`.

[34] Redis. `http://redis.io`.

[35] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Technical report, Google, 2010.

[36] SLEE, M., AGARWAL, A., AND KWIATKOWSKI, M. Thrift: Scalable cross-language services implementation. *Facebook White Paper 5*, 8 (2007).

[37] Snappy, a fast compressor/decompressor. `https://github.com/google/snappy`.

[38] spdlog: A Super fast C++ logging library. `https://github.com/gabime/spdlog`.

[39] Splunk. `https://www.splunk.com`.

[40] STALLMAN, R. M., MCGRATH, R., AND SMITH, P. *GNU Make: A Program for Directed Compilation*. Free software foundation, 2002.

[41] THE APACHE SOFTWARE FOUNDATION. Apache HTrace: A tracing framework for use with distributed systems. `http://htrace.incubator.apache.org`.

[42] THE APACHE SOFTWARE FOUNDATION. Apache HTTP Server Project. `http://httpd.apache.org`.

[43] THE APACHE SOFTWARE FOUNDATION. Apache Log4j 2. `https://logging.apache.org/log4j/log4j-2.3/manual/async.html`.

[44] THE APACHE SOFTWARE FOUNDATION. Apache Spark. `https://spark.apache.org`.

[45] THE APACHE SOFTWARE FOUNDATION. Log4j2 Location Information. `https://logging.apache.org/log4j/2.x/manual/layouts.html#LocationInformation`.

[46] VARDA, K. Protocol buffers: Googles data interchange format. *Google Open Source Blog, Available at least as early as Jul* (2008).

[47] YANG, S. NanoLog: an extremely performant nanosecond scale logging system for C++ that exposes a simple printf-like API. `https://github.com/PlatformLab/NanoLog`.

[48] Twitter Zipkin. `http://zipkin.io`.

[49] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on information theory 23*, 3 (1977), 337–343.

Model Governance: Reducing the Anarchy of Production ML

Vinay Sridhar   Sriram Subramanian   Dulcardo Arteaga
Swaminathan Sundararaman   Drew Roselli   Nisha Talagala
ParallelM

Abstract

As the influence of machine learning grows over decisions in businesses and human life, so grows the need for Model Governance. In this paper, we motivate the need for, define the problem of, and propose a solution for Model Governance in production ML. We show that through our approach one can meaningfully track and understand the who, where, what, when, and how an ML prediction came to be. To the best of our knowledge, this is the first work providing a comprehensive framework for production Model Governance, building upon previous work in developer-focused Model Management.

1 Introduction

Machine Learning (ML) and Deep Learning (DL) have recently made tremendous advances in algorithms, analytic engines, and hardware. However, production ML deployment is still nascent [17]. While production deployments are always challenging, ML generates unique difficulties [21, 7, 22]. We focus on the governance challenge: the management, diagnostic, compliance, and regulatory implications of production ML models. With recent demands for explainable/transparent ML [9, 3, 16, 5], the need to track provenance and faithfully reproduce ML predictions is even more serious. Given the strong data-dependent nature of ML/DL, even small changes in configurations can have unexpected consequences in predictions, making Governance critical to production ML.

Previous research has focused on *Model Management*: managing these models and enabling efficient reuse by developers [28, 25, 20]. Production deployment further complicates governance with i) complex topologies with retraining, ii) continuous inference programs that run in parallel with (re)training programs, iii) actions (such as model approvals) that need to be recorded for auditing, vi) model rollbacks that may occur in real time, and v) heterogeneous and distributed environments.

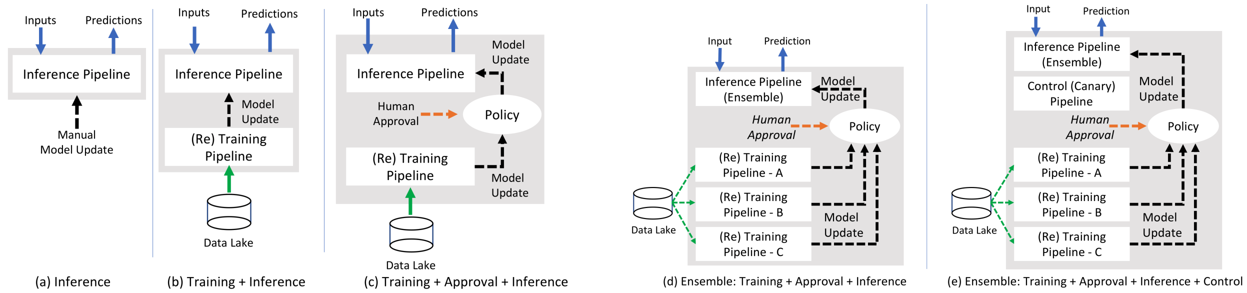
We define Production Model Governance as the ability to determine the creation path, subsequent usage, and consequent outcomes of an ML model, and the use of this information to accomplish a range of tasks including reproducing and diagnosing problems and enforcing compliance. In this paper, we propose and motivate a generic solution approach that can be adapted across different governance usage examples.

Our goal is to highlight the Model Governance problem and propose solutions. Our contributions are: i) we propose a definition for Production Model Governance and its necessary inclusive elements; ii) We propose a two-layer model for Governance. The lower layer contains each pipeline as a DAG and tracks everything (such as features, datasets, and code) similar to what is proposed by prior research. We add a second layer directed graph where each pipeline or policy invocation is a node and edges represent cross-pipeline, policy, and human action dependencies. We temporally track and correlate both of these levels to comprehensively cover Model Governance; iii) Using this model, we build a production Model Governance system that supports heterogeneous frameworks (currently, Spark, TensorFlow, Flink); iv) We propose a robust approach to a wide range of possible Governance applications via generic access to Governance metadata; and v) At the pipeline level, we expand upon prior research by illustrating a generic API-based instrumentation approach across analytic engines.

2 Motivation

Machine learning algorithms execute as ”pipelines”, which ingest data (via batch data lakes, stream brokers, etc.) and compute (feature engineering, model training, scoring, inference, etc.). Pipelines may run on engines (Spark, Flink, etc.) or as standalone programs.

To highlight the importance of Model Governance, we use an example medical application that leverages ML to recommend to a doctor which tests to run on a patient. The calling application sends user information to an ML prediction pipeline which returns a prediction. Figure 1 shows several examples of how this simple scenario can



(a) Inference (b) Training + Inference (c) Training + Approval + Inference (d) Ensemble: Training + Approval + Inference (e) Ensemble: Training + Approval + Inference + Control

Figure 1: **Evolution of ML Pipelines in production.** *(a) The prediction pipeline is executing in production and is using a model trained offline and uploaded. (b) A more dynamic scenario where the model itself is retrained on a schedule. (c) Yet another sophistication, where newly trained models undergo an approval process prior to production deployment. (d) Here an ensemble model is used for prediction (requiring each sub-model to be trained individually). Finally, (e) shows the scenario if a control pipeline (*canary*) is used in production to ensure that the primary prediction pipeline is behaving stably. The control pipeline could also be running a surrogate model to improve explainability.*

be put into production. While the basic function requires only one pipeline, once you add the need to improve accuracy via re-training, the need for human approvals, and state-of-the-art models, the complexity grows rapidly. We now illustrate sample Governance scenarios for the example in Figure 1(e).

Scenario 1: Say we needed to know why a certain patient was recommended a CT-Scan while another patient was not. For each recommendation, we would need to answer: Which model(s) were running in the ensemble? Which code was executing the models? When/How was each model trained (using which configurations and which features)? Which model provided the control pipeline and would its recommendation have differed? Which operator approved each of the models in the primary pipeline? Who approved the model in the control pipeline? Were any errors noted in this time frame? Can both predictions be reproduced in order to test for bias?

Scenario 2: Assume a data scientist wishes to leverage some of the models for a new production ML application. They would want to know under which circumstances the existing models were generated, as well as which datasets and features were used. These may also be required for production approval.

3 Model Governance

We define Production Model Governance as the ability to determine the creation path, subsequent usage, and consequent outcomes of an ML model, and the use of this information in various ways, as illustrated above. Given the wide range of usages, we believe any Model Governance solution should include:

Provenance/Lineage: For any ML prediction, the ability identify the exact sequence of events (datasets, trainings, code, pipelines, human approvals) that led to the event.

Reproducibility: The ability to replay the above sequence and replicate the prediction, thereby setting the context to investigate alternatives.

Audit and Compliance: The ability to evaluate all ML operations in an organization and determine compliance with regulations.

Leverage: The ability to reuse past ML work (such as algorithms, models, features) to determine whether the derived object is appropriate for the new usage.

Scale and Heterogeneity: The ability to work with many models, pipelines, and varied analytic engines and languages in a distributed setting such as Cloud/Edge.

Multiple Governance Metadata Usages: The ability to multi-purpose the metadata. For example, Data Scientists may analyze experiments or reuse models. Operators may diagnose issues, help address bias concerns, or ensure compliance to policies.

4 Approach and Design

Our solution must support simple to complex topologies, parallel pipelines, streaming and batch pipelines, pipelines changing state mid-run as new models arrive, policy actions and relationships between non-overlapping pipeline runs. For these we must provide: (i) Sufficient dependency information to infer provenance; (ii) Configuration including code and input parameters for reproducibility; (iii) A durable record of all metadata; (iv) Metadata from disjoint pipeline, possibly from different analytic engines and languages; (v) Metadata provenance, trends, and policy analysis and beyond.

4.1 The Intelligence Overlay Network

Core to our design is a two layer model we call the Intelligence Overlay Network (ION). An ION is a logical model that connects objects such as pipelines, policy execution modules, and messages between them. The first level of an ION is inspired by the traditional graph-based modeling of message passing parallel programs, where each node is a execution element and directed edges are messages between programs (allowing for cycles) [1]. In our case, execution nodes are ML pipelines or policy actions, and messages (such as ML models or events) are passed between them. At the second level of the ION, each execution node can itself be a DAG of components.

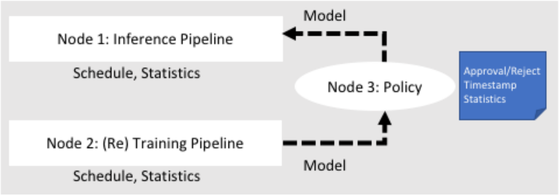


Figure 2: **ION for Flow in Figure 1(c)**

An ML pipeline is a single execution node in an ION. Figure 2 shows how Figure 1(c)'s pattern maps to an ION. The Inference and Training nodes are pipelines, the approval is a Policy node, and the edges show the path of a model. Figure 3 shows how the graph in Figure 2 is tracked by our system across time.

We apply this approach to ML workflows: (i) All execution elements are nodes. Nodes execute on a schedule (batch), continuously (streaming), or are event triggered; (ii) Nodes can pass messages and each message and send/receive events are recorded; (iii) Within each node, a DAG can be defined with its stages monitored.

The ION approach delivers important benefits. First, many ML pipelines map easily to the ION nodes as DAGs. Statistics from these pipelines are gathered as time-series variables that support Governance scenarios, for example, by providing required transparency to complex feature selection within a pipeline.

Second, the ION graph cleanly captures the dependencies and interactions between the pipelines, including their repeated executions over time, their connections to human actions, and their relationships to each other.

Finally, the combined statistics of both levels enables powerful usages, such as (a) tracking pipeline metrics like *confusion matrices* across multiple training runs, (b) comparing Models across multiple training runs, (c) tracking the approval actions of any human operator and cross-checking those against the performance of inference pipelines that ran the approved models.

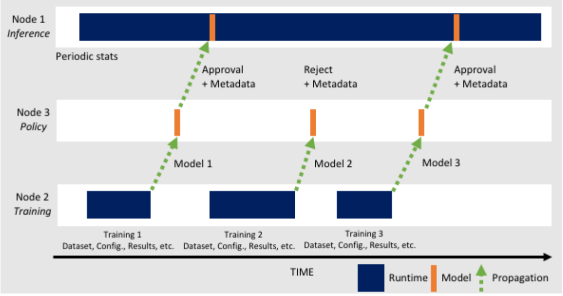


Figure 3: **ION Timeline**

4.2 System Design

We use an agent/server architecture. The agents run on each instance of an analytic engine (for example, a 15 node Spark cluster would have a single agent). Each agent communicates with its engine via standard interfaces. The server receives Governance metadata from the agents and interlinks the information via the ION. Currently our server also runs the policy execution code blocks. The server maintains a metadata database and manages garbage collection.

While recent Model Management approaches have been passive [28], we chose an active Agent based approach to enable disconnected operation (a common occurrence in Edge based ML environments [26]). During disconnections, the agent saves information locally and transmits it when connectivity is restored. If the agent is disconnected for long periods of time and runs out of local storage resources, some information can be lost. The agent/server architecture also enables scale and support of heterogeneous analytic engines.

This approach requires no changes to the analytic engines. We can also connect to existing analytic engines and can share analytic engines with other programs that we do not monitor. Any program that already works in these engines works in this environment. An Agent can also support custom standalone programs. The required changes to the programs themselves, for all the cases, are discussed in Section 4.3.

Figure 4 shows the database schema. The Level 1 schema includes tracked objects within a pipeline. The Level 2 schema captures the ION pattern as well as specific elements of each ION instance. Contextual information (such as which machines a particular pipeline ran on) is also captured. All objects are timestamped. All objects link to an ION and from the ION to each other.

4.3 Information Import and Export

ML pipelines exchange Governance metadata with our system in three ways (see Figure 5). First, a JSON-based ION definition which contains links to pipeline code is uploaded to our server. Second, each pipeline is instrumented via an API library to provide runtime metadata. Pipelines can export time series variables, digests, configuration, models, etc. Supported Model formats are PMML, SavedModel or opaque. Since our system also supports opaque model formats, other industry standard model formats (like ONNX) that we do not yet interpret can be immediately used. Third, for standard pipelines and models, we auto-extract metadata (like [25]).

The API library has import and export capabilities. On the export side, running pipelines send metadata to our Governance system. On the import side, running pipelines can query the database for stored metadata and perform additional analytics (see Section 4). The library implementation is engine-specific and to date we have implemented our API library for Spark, Flink, and Tensorflow and in Scala, Java, and Python.

Unlike prior approaches [25], we employ both a fully declarative approach (where the developer decides what to instrument) and automatic extraction wherever standardized pipelines and model formats allow. While train-

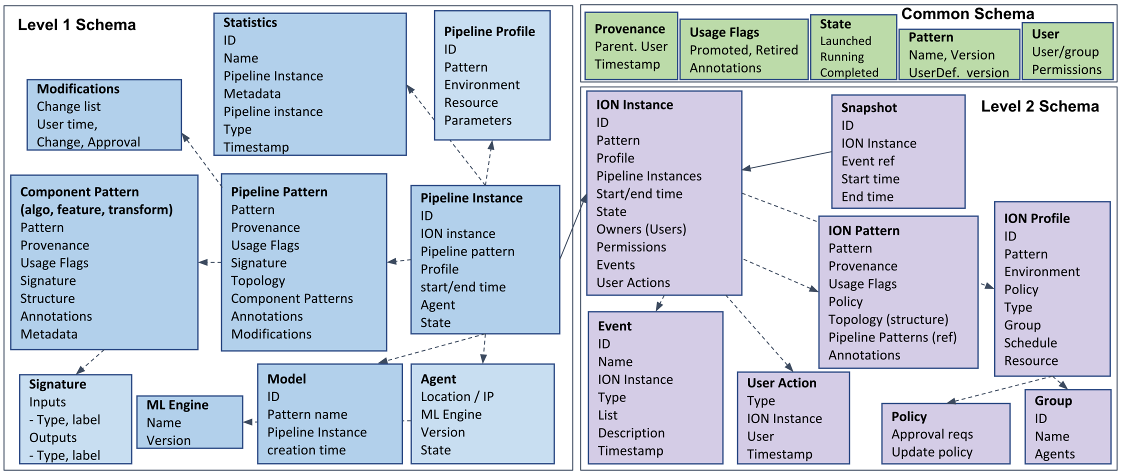
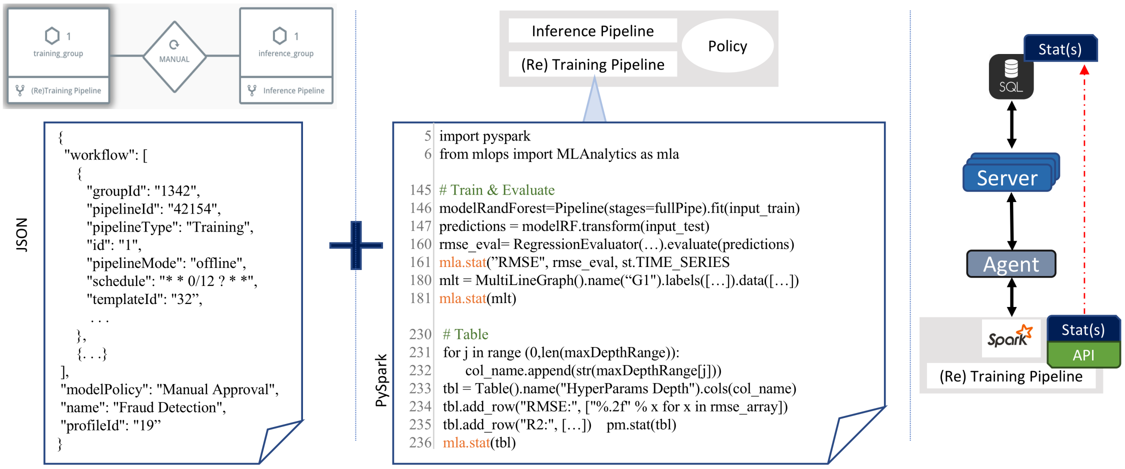


Figure 4: **Governance Schema**



```
{
 "workflow": [
   {
     "groupId": "1342",
     "pipelineId": "42154",
     "pipelineType": "Training",
     "id": "1",
     "pipelineMode": "offline",
     "schedule": "* * 0/12 ? * *",
     "templateId": "32”,
      . . .
   },
   {. . .}
 ],
 "modelPolicy": "Manual Approval",
 "name": "Fraud Detection",
 "profileId": "19”
}
```

```
import pyspark
from mlops import MLAnalytics as mla

# Train & Evaluate
modelRandForest=Pipeline(stages=fullPipe).fit(input_train)
predictions = modelRF.transform(input_test)
rmse_eval= RegressionEvaluator(…).evaluate(predictions)
mla.stat(”RMSE", rmse_eval, st.TIME_SERIES
mlt = MultiLineGraph().name(“G1").labels([…]).data([…])
mla.stat(mlt)

 # Table
 for j in range (0,len(maxDepthRange)):
     col_name.append(str(maxDepthRange[j]))
 tbl = Table().name("HyperParams Depth").cols(col_name)
 tbl.add_row("RMSE:", ["%.2f" % x for x in rmse_array])
 tbl.add_row("R2:", […])    pm.stat(tbl)
 mla.stat(tbl)
```

(a) ION Template

(b) Code w/ API Instrumentation

(c) Pipeline at Runtime

Figure 5: **Interaction of ML Pipelines with our Governance System**

ing pipelines can be quite structured (like SparkML), production inference pipelines have not standardized on any pipeline or code structure, running the gamut from auto-generated (via JPMML inference) to latency optimized hand-coded programs. This range of inference pipeline structures renders approaches that rely only on automatic extraction impractical for generalized production usage. With our approach, even custom written standalone programs can be instrumented for Governance.

4.4 Correlation and Causality

IONs are used to create a coherent time view of all metadata by (a) merging the views of multiple concurrent pipelines or policy blocks and (b) by relating different executions of the same logical node (e.g., re-training).

Causality: Within each ION node, we assume monotonically increasing timestamps. Across nodes within an ION, causality is established via messages.

Inter-Node Correlation: Across ION nodes, we assume correlation based on synchronized timestamps. Provenance is derived directly from messages and not from time correlation. Correlation is only used if a single sequence of statistics from the ION is retrieved for visualization. We have not used logical clocks to date because the governance usages we have worked with can be addressed sufficiently with the approaches above.

4.5 Governance Usages

Model Governance metadata has multiple uses. Any pipeline can use the API library to extract and compute additional analytics on governance metadata. For example, a pipeline can analyze all models approved by a user and correlate with training accuracy metrics or do additional model selection or model improvement. Compliance enforcement can be done via functions that periodically review recent metadata and confirm adherence to specific polices (such as human model approval or training accuracy thresholds for production deployed models). The metadata information can be used for advanced optimizations such as Meta-Learning. The applications of this approach are vast and to date we have only used our system to explore simple usages. Due to lack of space, we do not elaborate on this capability further in this paper but it is an area of our future work.

Object	Occurrence	Count	Size
Models	multiple	24	48KB
Statistics	multiple	536K	242MB
Signatures	multiple	1.8K	136MB
Pipelines	once	3	14KB
Node	once	3	1KB
Events	multiple	84	140KB
Logs	multiple	264	74MB
Total	N/A	5̃38K	452MB

Table 1: **Governance information captured from the three node ION after a day of execution.**

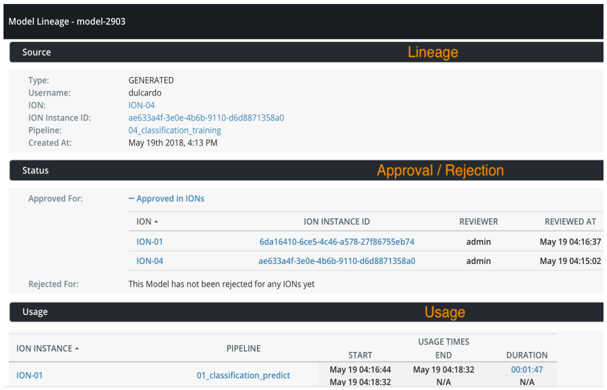
5 Evaluation

A governance solution should include lineage, reproducibility, audit, leverage, scale, heterogeneity, and multiple usages (see Section 3). We conduct a simple experiment to illustrate lineage, audit, scale, and heterogeneity in our solution. Additionally, we measure the overheads of Governance metadata gathering, including instrumentation and model propagation.

Our experiment consists of a three node ION: a Spark training pipeline, a Flink inference pipeline, and a policy module configured to always approve models (Figure 1c). The training pipeline is batch Logistic Regression with hyperparameter search in PySpark training a new model every hour. The selected model propagates via the Policy node to the Flink streaming inference in Scala. Governance metadata gathered includes input and prediction signatures (histograms), ML statistics, code, libraries, etc. ML statistics are any items reported via the API, including generic metrics such as accuracy, precision, recall, etc. and algorithm-specific metrics. Hyperparameter search configuration and result information is also reported via the API.

Figure 6 shows the dashboard view of a model generated in the ION. The governance dashboard provides the lineage information for every model generated (or uploaded), the training pipeline and the parameters used to generate the selected model, model approval information and the subsequent usage of the selected model in inference pipelines. Each hyperlink within the dashboard provides in-depth information on ION template, pipeline, configuration, statistics, etc. These in-depth views are not included due to lack of space.

Table 1 summarizes the list of objects that are captured in our system during the ION's execution. The overheads of our API instrumentation was less then 1% in this experiment and model propagation delay on average was around 3s for models of size 2KB. We were able to both replay and reproduce predictions using our system.



Model Lineage - model-2903

Source

Type: GENERATED
Username: dulcardo
ION: ION-04
ION Instance ID: ae633a4f-3e0e-4b6b-9110-d6d8871358a0
Pipeline: 04_classification_training
Created At: May 19th 2018, 4:13 PM

Status

Approved For: – Approved in IONs

ION ▴	ION INSTANCE ID	REVIEWER	REVIEWED AT
ION-01	6da16410-6ce5-4c46-a578-27f86755eb74	admin	May 19 04:16:37
ION-04	ae633a4f-3e0e-4b6b-9110-d6d8871358a0	admin	May 19 04:15:02

Rejected For: This Model has not been rejected for any IONs yet

Usage

ION INSTANCE ▴	PIPELINE	USAGE TIMES START	USAGE TIMES END	USAGE TIMES DURATION
ION-01	01_classification_predict	May 19 04:16:44 May 19 04:18:32	May 19 04:18:32 N/A	00:01:47 N/A

Figure 6: **Model Governance Page:** This page highlights the lineage, approval/rejection status across IONs, and usage of the selected model across IONs.

6 Related Work

The related work closest to ours includes [28, 25, 20]. We add several fundamental contributions over each. First we define and implement the ION model which links multiple pipelines and policy actions, including a full governance metadata and schema. Second, while both we and [25] integrate with multiple analytic engines, our approach adds a fully declarative instrumentation approach to the base auto-extraction approach, enabling our system to work with changing analytic engines and standalone ML programs. Finally, we have designed and built an import/export function where programs can access and compute additional analytics on Governance metadata, enabling a vast range of Governance usages which can themselves be tracked and managed. Additional model management approaches have been presented in [18]. Production ML challenges have been described in [21, 7, 22, 24, 4, 15, 8]. Meta-learning is discussed in [23, 13]. Overviews of analytic engines, libraries, and model serving systems are in [19, 10, 11, 12, 27, 2, 29, 6, 14].

7 Discussion and Future Work

Our system addresses most of the goals laid out in Section 3. We can track any prediction (or other event) to all related pipelines, datasets, execution configurations, code and human actions, and reproduce the functional steps. Via both tracking of input dataset pathnames and dataset signatures, we identify which data was used to generate which model. Using the API library, a user can write a pipeline to extract and analyze any information in the database to evaluate compliance or do audits. As another example, programs using our API can analyze data trends and correlate them with model outcomes.

For future work, we plan to explore the possibilities of meta-learning via Governance metdata and expand on how Governance relates to AI explainability. We want to couple the provenance data with explainable ML to help answer the question of not just how/where/what/when but also why ML decisions were made.

References

[1] The message passing parallel programming model, 1995.

[2] Apache. flink. Internet draft, 2017.

[3] Gdpr online, 2017. http://www.eudgpr.org.

[4] Michelangelo: Uber's machine learning platform. Internet draft, 2017.

[5] New york bill on algorithmic bias, 2017. http://www.businessinsider.com/algorithmic-bias-accountability-bill-passes-in-new-york-city-2017-12.

[6] ABADI, M., AND ET AL. Tensorflow: Large-scale machine learning on heterogeneous systems. Internet draft, 2015.

[7] BAYLOR, D., BRECK, E., CHENG, H.-T., FIEDEL, N., FOO, C. Y., HAQUE, Z., HAYKAL, S., ISPIR, M., JAIN, V., KOC, L., KOO, C. Y., LEW, L., MEWALD, C., MODI, A. N., POLYZOTIS, N., RAMESH, S., ROY, S., WHANG, S. E., WICKE, M., WILKIEWICZ, J., ZHANG, X., AND ZINKEVICH, M. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2017), KDD '17, ACM, pp. 1387–1395.

[8] BRECK, E., CAI, S., NIELSEN, E., SALIB, M., AND SCULLEY, D. Whats your ml test score? a rubric for ml production systems. reliable machine learning in the wild. *Neural Information Processing Systems (NIPS) Workshop.* (2016).

[9] BURT, A. Is there a 'right to explanation' for machine learning in the gdpr. Internet draft, 2017.

[10] CHEN, T., LI, M., LI, Y., LIN, M., WANG, N., WANG, M., XIAO, T., XU, B., ZHANG, C., AND ZHANG, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems.

[11] CRANKSHAW, D., BAILIS, P., GONZALEZ, J. E., LI, H., ZHANG, Z., FRANKLIN, M. J., GHODSI, A., AND JORDAN, M. I. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *CoRR abs/1409.3809* (2014).

[12] CRANKSHAW, D., WANG, X., ZHOU, G., FRANKLIN, M. J., GONZALEZ, J. E., AND STOICA, I. Clipper: A low-latency online prediction serving system. *CoRR abs/1612.03079* (2016).

[13] FEURER, M., SPRINGENBERG, J. T., AND HUTTER, F. Initializing bayesian hyperparameter optimization via meta-learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (2015), AAAI'15, AAAI Press, pp. 1128–1135.

[14] FIEDEL, N. Serving models in production with tensorflow serving, 2017.

[15] FLAUONAS, I. Beyond the technical challenges for deploying machine learning solutions in a software company. Internet draft, 2017.

[16] GARFINKEL, S., MATTHEWS, J., SHAPIRO, S., AND SMITH, J. Towards algorithmic transparency and accountability. *Communications of the ACM, Vol. 60 No. 9, Page 5* (2016).

[17] INSTITUE, M. G. Artificial intelligence: The next digital frontier?, 2017.

[18] KUMAR, A., MCCANN, R., NAUGHTON, J., AND PATEL, J. M. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Rec. 44*, 4 (May 2016), 17–22.

[19] MENG, X., BRADLEY, J., YAVUZ, B., SPARKS, E., VENKATARAMAN, S., LIU, D., FREEMAN, J., TSAI, D., AMDE, M., OWEN, S., XIN, D., XIN, R., FRANKLIN, M. J., ZADEH, R., ZAHARIA, M., AND TALWALKAR, A. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res. 17*, 1 (Jan. 2016), 1235–1241.

[20] MIAO, H., LI, A., DAVIS, L. S., AND DESHPANDE, A. Towards unified data and lifecycle management for deep learning. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)* (April 2017), pp. 571–582.

[21] PIERRE ANDREWS, ADITYA KALRO, H. M. A. S. Productionizing machine learning pipelines at scale. *Machine Learning Systems Workshop at ICML.* (2016).

[22] POLYZOTIS, N., ROY, S., WHANG, S. E., AND ZINKEVICH, M. Data management challenges in production machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 1723–1726.

[23] Schelter, S., Soto, J., Markl, V., Burdick, D., Reinwald, B., and Evfimievski, A. Efficient sample generation for scalable meta learning. In *2015 IEEE 31st International Conference on Data Engineering* (April 2015), pp. 1191–1202.

[24] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. Hidden technical debt in machine learning systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2* (Cambridge, MA, USA, 2015), NIPS'15, MIT Press, pp. 2503–2511.

[25] Sebastian Schelter, Joos-Hendrik Boese, J. K. T. K. S. S. Automatically tracking metadata and provenance of machine learning experiments. *Workshop on ML Systems at NIPS (MLSys) Workshop.* (2017).

[26] Talagala N., Sundararaman S., S. V. A. D. L. Q. S. S. G. S. R. D. K. S. Eco: Harmonizing edge and cloud with ml/dl orchestration. In *Proceedings of USENIX HotEdge 2018* (2018).

[27] Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. Openml: networked science in machine learning. *CoRR abs/1407.7722* (2014).

[28] Vartak, M., Subramanyam, H., W-E., L., Viswanathan, S., and Madden, S. Demonstration of modeldb: a system for managing machine learning models. *Neural Information Processing Systems (NIPS) Workshop.* (2016).

[29] Zaharia, M., Xin, R. S., Wendell, P., Das, T., A. M. D. A. M. X. R. J. V. S. F. M. G. A. G. J. S. S., and Stoica, I. Apache spark: a unified engine for big data processing. *Commun. ACM 59* (2016), 56–65.

Fine-grained consistency for geo-replicated systems

Cheng Li[♭], Nuno Preguiça[‡], Rodrigo Rodrigues[*]
[♭]*University of Science and Technology of China,* [‡]*NOVA LINCS & FCT, Univ. NOVA de Lisboa,*
[*]*INESC-ID & Instituto Superior Técnico, Universidade de Lisboa*

Abstract

To deliver fast responses to users worldwide, major Internet providers rely on geo-replication to serve requests at data centers close to users. This deployment leads to a fundamental tension between improving system performance and reducing costly cross-site coordination for maintaining service properties such as state convergence and invariant preservation. Previous proposals for managing this trade-off resorted to coarse-grained operations labeling or coordination strategies that were oblivious to the frequency of operations. In this paper, we present a novel fine-grained consistency definition, Partial Order-Restrictions consistency (or short, PoR consistency), generalizing the trade-off between performance and the amount of coordination paid to restrict the ordering of certain operations. To offer efficient PoR consistent replication, we implement Olisipo, a coordination service assigning different coordination policies to various restrictions by taking into account the relative frequency of the confined operations. Our experimental results show that PoR consistency significantly outperforms a state-of-the-art solution (RedBlue consistency) on a 3-data center RUBiS benchmark.

1 Introduction

To cope with the demand for fast response times [36] from an increasingly large user base, many Internet service providers such as Google [8], Microsoft [9], Facebook [13] or Amazon [3] replicate data across multiple geographically dispersed data centers [38, 21, 20, 2]. However, geo-replication also leads to an inherent tension between achieving high performance and ensuring properties such as state convergence (i.e., all replicas eventually reach the same final state) and invariant preservation (i.e., the behavior of the system obeys its specification, which can be defined as a set of application-specific invariants to be preserved) [22, 40, 30, 17, 16].

Some proposals address this fundamental tension in geo-replication by weakening strong consistency to different extents: some researchers suggest to completely drop strong consistency and instead adopt some form of weaker consistency such as eventual consistency [22, 41, 19] or causal consistency [32]; other approaches allow multiple consistency levels to coexist in a single system [30, 17, 12, 4]. As an example of the latter group, our prior proposal on RedBlue consistency [30], allows some operations to execute under strong consistency (and therefore incur a high performance penalty) while other operations can execute under weaker consistency (namely causal consistency). The core of this solution is a labeling methodology for guiding the programmer to assign consistency levels to operations. The labeling process works as follows: operations that either do not commute w.r.t. all others or potentially violate invariants must be strongly consistent, while the remaining ones can be weakly consistent.

This binary classification methodology is effective for many applications, but it can also lead to unnecessary coordination in some cases. In particular, as we will later illustrate, there are cases where it is important to synchronize the execution of two specific operations, but those operations do not need to be synchronized with any other operation in the system (and this synchronization would happen across all strongly consistent operations in the previous scheme). Furthermore, while concepts such as *conflict relation* in generic broadcast [34] and *token* by Gotsman et al. [24] allow for a finer-grained coordination of operations, these either lack a precise method for identifying a set of restrictions to ensure safety or an implementation that achieves efficient coordination by adapting to the observed workload.

To overcome these limitations, in this paper, we propose a novel generic consistency definition, *Partial Order-Restrictions consistency* (or short, *PoR consistency)*, which takes a set of restrictions as input and forces these restrictions to be met in all partial orders. This creates the opportunity for defining many consistency guarantees within a single replication framework by expressing consistency levels in terms of visibility restrictions on pairs of operations. Weakening or strengthening the consistency semantics is achieved by imposing fewer or more restrictions.

Under PoR consistency, the key to making a geo-replicated deployment of a given application perform well is to identify a set of restrictions over pairs of its operations so that state convergence and invariant preservation are ensured if these restrictions are enforced throughout all executions of the system. However, this is challenging because missing required restrictions may cause applications to diverge state or violate invariants, while placing unnecessary restrictions will lead to a performance penalty due to the additional coordination. To this end, we design principles guiding programmers to identify the important restrictions while avoiding unnecessary ones.

Furthermore, from a protocol implementation perspective, given a set of restrictions over pairs of operations, there exist several coordination protocols that can be used for enforcing a given restriction, such as Paxos, distributed locking, or escrow techniques. However, depending on the frequency over time with which the system receives operations confined by a restriction, different coordination approaches lead to different performance trade-offs. Therefore, to minimize the runtime coordination overhead, we also propose an efficient coordination service called Olisipo that helps replicated services use the most efficient protocol by taking into account the system workload.

To demonstrate the power of PoR consistency, we extended RUBiS to incorporate a closing auction functionality, determined how to best run it under PoR consistency, replicated it with Olisipo, and compared its performance against a RedBlue consistent version. Our experimental results show that PoR consistency requires fewer restrictions and offers a significantly better performance than RedBlue consistency.

2 Preliminaries

2.1 System model

We assume a geo-distributed system with state fully replicated across k sites denoted by $site_0 \dots site_{k-1}$, where each site hosts a replica, and each replica runs as a deterministic state machine. In the rest of the document, the terms “site” and “replica” are interchangeable.

The system defines a set of operations $\mathscr{U}$ manipulating a set of reachable states $\mathscr{S}$. Each operation u is initially submitted by a user at one site which we call u’s *primary site* and denote $site(u)$. An operation is defined mathematically as a function that receives the current state of the system and returns another function corresponding to its side effects. We refer to the former function as the *generator* function, denoted by g_u; this generator function, when applied to a given state $S \in \mathscr{S}$, returns a *shadow* function or *shadow operation*, denoted $h_u(S)$.

Implementation-wise, the generator function will first execute in a *sandbox* against the current state of the replica at the primary site, without interference from other concurrent operations. In this phase, the execution only identifies what changes u would introduce to state S that is observed by u and will not commit these changes. At the end of executing g_u, the identified side-effect or shadow operation $h_u(S)$ will be sent and applied across all replicas including the primary site.

A desirable property is that all replicas that have applied the same set of shadow operations are in the same state, i.e., the underlying system offers **state convergence**. In addition, the system maintains a set of application-specific **invariants**. For instance, an online shopping service cannot sell more items than those available in stock. To capture this notion, we define the function $valid(S)$ to be *true* if state S satisfies all these invariants and *false* otherwise.

2.2 RedBlue consistency

Our prior proposal called RedBlue consistency [30] is based on a division of shadow operations into blue operations, whose order of execution can vary from site to site, and red operations that must execute in the same relative order at all sites. For guiding developers in making use of RedBlue consistency, this work identified that a condition for ensuring state convergence is that a shadow operation must be labeled red if it is not globally commutative. For ensuring that invariants are maintained, a sufficient condition was identified, stating that all shadow operations that may violate an invariant when being applied against a different state from the one they were generated must be labeled red. For the remaining shadow operations, which have passed the two condition checks, we can safely label them blue.

3 Partial Order-Restrictions consistency

3.1 Motivating example

We illustrate the limitations of coarse-grained labeling schemes like RedBlue consistency through an eBay-like auction service in Fig.1, where an operation `placeBid` (Fig.1(a)) creates a new bid for an item if the corresponding auction is still open, and an operation `closeAuction` (Fig.1(c)) closes an auction and declares a single winner. In this example, the application-specific invariant is that the winner must be associated with the highest bid across all accepted bids. The other two subfigures (Fig.1(b) and Fig.1(d)) depict the commutative shadow operations of these two operations.

When applying RedBlue consistency to replicate such an auction service, we note that the concurrent execution under weak consistency of a `placeBid` with a bid higher than all accepted bids and a `closeAuction` can lead to the violation of the application invariant. This happens because the generation of `closeAuction’` will ignore the highest bid created by the concurrent shadow `placeBid’`. Unfortunately, the only way to address this issue in RedBlue consistency is to label both shadow operations as strongly consistent, i.e., all shadow operations of either type will be totally ordered w.r.t each other, which will incur a high overhead in geo-distributed settings. Intuitively, however, there is no need to order pairs of `placeBid’` shadow operations, since a bid coming before or after another does not affect the winner selection. This highlights that a coarse-grained operation clas-

```
boolean placeBid(int itemId, int clientId, int bid){
 boolean result = false;
 beginTxn();
 if(open(itemId)){
   createShadowOp(placeBid', itemId, clientId, bid);
   result = true;
 }
 commitTxn();
 return result;
}
```

(**a**) Original placeBid operation.

```
placeBid'(int itemId, int clientId, int bid){
  exec(INSERT INTO bidTable VALUES (bid, clientId, itemId));
}
```

(**b**) Shadow placeBid' operation.

```
int closeAuction(int itemId){
 int winner = -1;
 beginTxn();
 close(itemId);
 winner = exec(SELECT userId FROM bidTable WHERE iId = itemId
       ORDER BY bid DESC limit 1);
 createShadowOp(closeAuction', itemId, winner);
 commitTxn();
 return winner;
}
```

(**c**) Original closeAuction operation.

```
closeAuction'(int itemId, int winner){
 close(itemId);
 exec(INSERT INTO winnerTable VALUES (itemId, winner));
}
```

(**d**) Shadow closeAuction' operation.

Figure 1: Pseudocode for the `placeBid` and `closeAuction` operations of an auction site

sification into two levels of consistency can be conservative, and some services could benefit from additional flexibility in terms of the level of coordination.

To overcome these limitations of RedBlue consistency, we next propose Partial Order-Restrictions consistency (or short, PoR consistency), a novel consistency model that allows the developer to reason about various fine-grained consistency requirements in a single system. The key intuition behind our proposal is that this model is generic and can be perceived as a set of restrictions imposed over admissible partial orders across the operations of a replicated system.

3.2 Defining PoR consistency

The definition of PoR consistency includes three important components: (1) a set of restrictions, which specify the visibility relations between pairs of operations; (2) a restricted partial order (or short, R-order), which establishes a (global) partial order of operations respecting operation visibility relations; and (3) a set of site-specific causal serializations, which correspond to total orders in which the operations are locally applied. We define these components formally as follows:

Definition 1 (Restriction). *Given a set of operations U, a* restriction *is a symmetric binary relation on $U \times U$.*

For any two operations u and v in U, if there exists a restriction relation between them, we denote this relation as $r(u,v)$.

Definition 2 (Restricted partial order). *Given a set of operations U, and a set of restrictions R over U, a* restricted partial order *(or short,* R-order*) is a partial order $O = (U, \prec)$ with the following constraint: $\forall u, v \in U$, $r(u,v) \in R \implies u \prec v \vee v \prec u$.*

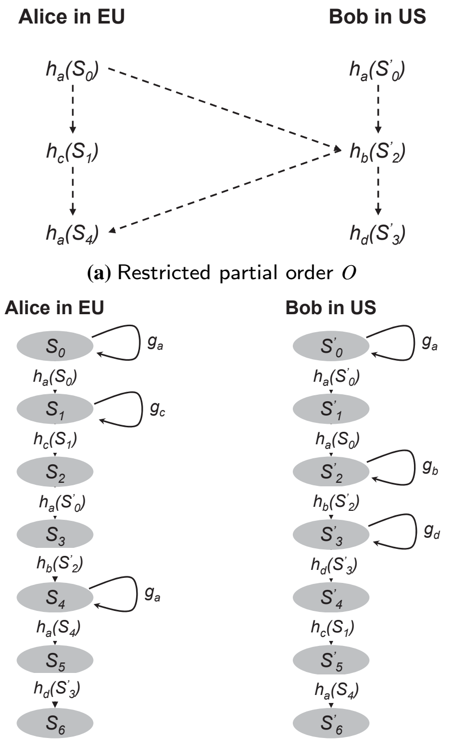
We say that the restrictions in R are met in the corresponding R-order if this order satisfies the above definition. This definition places constraints on a global view of a replicated system; however, it fails to explain how each individual replica at every site will behave according to this global view. When user requests are accepted by any site, that site executes their generator operations and creates corresponding shadow operations which will be replicated across all sites. In addition, every site not only commits shadow operations created by itself, but also applies remote ones shipped from all other sites against its local state. We denote U as the set of shadow operations produced across all sites, while for a site i, we denote V_i as its generator operation set. The following definition models the execution of each site as a growing linear extension of the global R-order, which incorporates a notion of causality, due to the fact that the visibility dependencies that are established when shadow operations are initially generated, are then preserved while the corresponding shadow operations are replicated.

Definition 3 (Causal legal serialization). *Given a site i, an R-order $O = (U, \prec)$ and the set of generator operations V_i received at site i, we say that $O_i = (U \cup V_i, <_i)$ is an* i-causal legal serialization *(or short, a* causal serialization*) of O if*

- *O_i is a total order;*
- *$(U, <_i)$ is a linear extension of O;*
- *For any $h_v(S) \in U$ generated by $g_v \in V_i$, (1) S is the state obtained after applying the sequence of shadow operations preceding g_v in O_i; (2) For any $h_u(S') \in U$, $h_u(S') <_i g_v$ in O_i iff $h_u(S') \prec h_v(S)$ in O.*

Definition 4 (Partial Order-Restrictions consistency). *A replicated system $\mathscr{S}$ spanning k sites with a set of restrictions R is* Partial Order-Restrictions consistent *(or short,* PoR consistent*) if each site i applies shadow operations according to an i-causal serialization of R-order O.*

Fig.2 shows a restricted partial order and its causal legal serializations executed at two sites, namely EU and US, where we restrict pairs of shadow operations where one corresponds to a and the other to b. When the US site executes a generator of b, g_b, it realizes that the shadow operation it would generate may need to be re-



(a) Restricted partial order O

(b) Causal legal serializations of O

Figure 2: Restricted partial order of shadow operations and its causal legal serializations for a system spanning two sites. There exists a restriction $r(h_a(S), h_b(S))$ for all valid S. Dotted arrows in Fig.2a indicate dependencies between shadow operations. Loops in Fig.2b represent generator operations.

stricted w.r.t a concurrent shadow operation initially triggered at the EU site. As a result, g_b at the US site must wait until the respective concurrent shadow operation $h_a(S_0)$ gets propagated from Alice's site to Bob's site. Then g_b will read the state introduced by locally applying $h_a(S_0)$ from Alice, and produce a shadow operation $h_b(S'_2)$. Note that this production will establish a dependency between $h_a(S_0)$ and $h_b(S'_2)$ (as shown in Fig.2a), thus enforcing that they cannot be applied in different relative orders in all causal legal serializations (as shown in Fig.2b). Unlike these two shadow operations, we do not restrict any pair of shadow operations of a; as such, the first operations issued by both Alice and Bob will be concurrently executed without being aware of each other. This example indicates the flexibility and performance benefits of having PoR consistency, compared with RedBlue consistency, since under the latter model all shadow operations of a and b would be serialized w.r.t each other.

4 Restriction inference

When replicating a service under PoR consistency, the first step is to infer restrictions to ensure two important system properties, namely state convergence and invariant preservation. The major challenge we face is to identify a minimal set of restrictions for making the replicated service converge and not violate invariants. With regard to state convergence, we take a similar methodology adopted in prior research [37, 30, 29], which is to check operation commutativity.

To preserve application-specific invariants, instead of totally ordering all *non-invariant safe* shadow operations, i.e., those that potentially transition from a valid state to an invalid one, we try to identify a minimal set of shadow operations that lead to an invariant violation when they are running concurrently in a coordination-free manner. By minimal, we mean that removing any operation from that set would no longer meet that goal. Once this set is identified, adding a restriction between any pair of its operations is sufficient to eliminate the problematic executions.

4.1 State convergence

A PoR consistent replicated system is state convergent if all its replicas reach the same final state when the system becomes quiescent, i.e., for any pair of causal legal serializations of any R-order, L_1 and L_2, we have $S_0(L_1) = S_0(L_2)$, where S_0 is a valid initial state. We state a necessary and sufficient condition to achieve this in the following theorem.

Theorem 5. *A PoR consistent system $\mathscr{S}$ with a set of restrictions R is* ***convergent****, if and only if, for any pair of its shadow operations u and v, $r(u, v) \in R$ if u and v don't commute.*[1]

Unlike RedBlue consistency, under which all operations that are not globally commutative must be totally ordered, PoR consistency only requires that an operation must be ordered w.r.t another one if they do not commute.

4.2 Invariant preservation

In RedBlue consistency, the methodology for identifying restrictions imposed on RedBlue orders for maintaining invariants is to check if a shadow operation is *invariant safe* or not (meaning whether it can potentially violate invariants when executed against a different state from the one that it was generated from). If not, to avoid invariant violations, the generation and replication of all non-invariant safe shadow operations must be coordinated. However, we observed that for some non-invariant safe shadow operations u, the corresponding violation only happens when a particular subset of non-invariant safe shadow operations (including u) are not partially ordered. Therefore, to eliminate all invariant violating executions with a minimal amount of coordination, we need to precisely define, for each violation, the minimal set of non-invariant safe shadow operations that are involved.

[1] All proofs are in a separate technical report [7].

We call this set an `invariant-conflict operation set`, or short, `I-conflict set`. Preserving invariants only requires adding a single restriction over any two shadow operations from each `I-conflict set` so that the concurrent violating executions will be eliminated from all admissible partial orders. We formally define `I-conflict sets` as follows.

Definition 6 (Invariant-conflict operation set)**.** *A set of shadow operations G is an* invariant-conflict operation set *(or* I-conflict set*) if the following conditions are met:*

- *$\forall u \in G$, u is non-invariant safe;*
- *$|G| > 1$;*
- *$\forall u \in G$, $\forall$ sequence P consisting of all shadow operations in G except u, i.e., $P = (G \setminus \{u\}, <)$, $\exists$ a reachable and valid state S, s.t. $S(P)$ is valid, and $S(P+u)$ is invalid.*

In the above definition, the last point asserts that G is minimal, i.e., removing one shadow operation from it will no longer lead to invariant violations. We will use the following example to illustrate the importance of minimality. Imagine that we have an auction on an item i being replicated across three sites such as US, UK and DE, and having initially a 5 dollar bid from *Charlie*. Suppose also that three shadow operations, namely, $placeBid'(i,\ Bob,\ 10)$, $placeBid'(i,\ Alice,\ 15)$, and $closeAuction'(i)$ are accepted concurrently at the three locations, respectively. After applying all of them against the same initial state at every site, we end up with an invalid state, where *Charlie* rather than *Bob* and *Alice* won the auction. This invariant violating execution involves three concurrent shadow operations, but one of the two bid placing shadow operations is not necessary to be included in G, as even after excluding the request from either *Bob* or *Alice*, the violation still remains. This is reflected in Definition 6, according to which $\{placeBid', closeAuction'\}$ is an `I-conflict set`, while $\{placeBid', placeBid', closeAuction'\}$ is not. Intuitively, avoiding invariant violations requires preventing all operations from the `I-conflict set` from running in a coordination-free manner. The minimality property enforced in the `I-conflict set` definition allows us to avoid adding unnecessary restrictions.

Based on the above definition, we formulate the invariant preservation property into the following theorem.

Theorem 7. *Given a PoR consistent system $\mathscr{S}$ with a set of restrictions $R_{\mathscr{S}}$, for any execution of $\mathscr{S}$ that starts from a valid state, no site is ever in an invalid state, if the following conditions are met:*

- *for any of its `I-conflict set` G, there exists a restriction $r(u,v)$ in $R_{\mathscr{S}}$, for at least one pair of shadow operations $u,v \in G$; and*
- *for any pair of shadow operations u and v, $r(u,v)$ in $R_{\mathscr{S}}$ if u and v do not commute.*

Algorithm 1 Find state convergence restrictions

```
1: function SCRDISCOVER($T$)        ▷ $T$: the set of shadow
   operations of the target system
2:     $R \leftarrow \{\}$                  ▷ $R$: the restriction set
3:     for $i \leftarrow 0$ to $|T|-1$ do
4:         for $j \leftarrow i$ to $|T|-1$ do
5:             if $T_i$ do not commute with $T_j$ then
6:                 $R \leftarrow R \cup \{r(T_i, T_j)\}$
7:     return $R$
```

Algorithm 2 Find invariant preserving restrictions

```
function IPRDISCOVER($T$)
    $R \leftarrow \{\}$                      ▷ $R$: the restriction set
    $Q \leftarrow$ power set of $T$
    for all $Q' \in Q$ do
        if ICONFLICTCHECK($Q'$) then
            if $|Q'| == 1$ then
                $R \leftarrow R \cup \{r(Q'_0, Q'_0)\}$
            else if $\forall u,v \in Q', r(u,v) \notin R$ then
                $R \leftarrow R \cup \{r(u,v)\}$, for an arbitrary
choice of $u,v \in Q'$
    return $R$
function ICONFLICTCHECK($T$)
    if $|T| == 1$ then
        if $\neg(T_0.post \implies T_0.wpre)$ then
            return true
    if $|T| > 1$ then
        $subset\_iconflict \leftarrow$ false
        for $i \leftarrow 2$ to $|T|-1$ do
            for all $R$ s.t. $|R| == i$ and $R \subset T$ do
                if ICONFLICTCHECK(R) then
                    $subset\_iconflict \leftarrow$ true
                    break
        if $!subset\_iconflict$ then
            for all $t \in T$ do
                $post \leftarrow \wedge_{x \in T \setminus \{t\}} x.post$
                if $\neg(post \implies t.wpre)$ then
                    return true
    return false
```

4.3 Identifying restrictions

The key to striking a sensible balance between performance and consistency semantics is to identify a minimal set of restrictions that ensure both state convergence and invariant preservation. With regard to the former property, inspired by Theorem 5, we design a discovery method for finding restrictions to ensure state convergence (Alg. 1). This method systematically performs an operation commutativity analysis between pairs of shadow operations: if two shadow operations do not commute, then a restriction between them is added to the returning restriction set (line 5-6).

To discover restrictions for preserving invariants, we could exhaustively explore all `I-conflict` sets consisting of concurrent shadow operations that trigger violations. However, it is very challenging to achieve this

```
//each permission consists of a set of operations
Permission p;

//receive a set of operations that need to be monitored
Permission getPermission(TxnId tid, String opName);

//wait until the set of operations in p have been applied
void waitForBeingExecuted(TxnId tid, Permission p);

//clean up all required resources occupied
void cleanUp(TxnId tid);
```

Figure 3: Olisipo coordination policy interface

since there might exist a large number of violating executions containing at least one I-conflict set. To make this exploration more efficient, we first collapse many similar executions of a replicated system into a single execution class, and then perform a weakest precondition and postcondition analysis over these classes [23].

In particular, for every shadow operation u, we denote $u.wpre$ as its weakest precondition, a condition on the initial state ensuring that u always preserves invariants. We also denote $u.post$ as the postcondition that captures the side effects of operations through a condition that always holds after the operation is executed. In Alg. 2, we flag a set of shadow operations T as `I-conflicting` if either of the following two conditions is met: (a) T contains a single operation t and t is self-conflicting, i.e., $t.wpre$ is invalidated by $t.post$ (line 12-14); or (b) $|T| > 1$, any subset of T is not `I-conflicting` (but can be self-conflicting) and there exists an operation u from T such that $u.wpre$ can be invalidated by the compound postcondition of all the operations in $T \setminus \{u\}$ (line 16-26).

Once these `I-conflict` sets are determined, then for each such set T, we add a restriction between an arbitrary pair of shadow operations from T if no pair of operations from that set was previously restricted (line 8-9). Otherwise, T will be skipped since the preexisting restriction suffices to preserve invariants. In addition, for shadow operations that are self-conflicting, we have to place a restriction between pairs of shadow operations of that type (line 6-7).

5 Design and Implementation of Olisipo

Several coordination protocols can be used for enforcing a given restriction, such as Paxos, distributed locking, or escrow techniques. However, depending on the observed runtime frequency of operations confined by a restriction, different approaches lead to different performance.

In the previously mentioned auction example, maintaining the invariant that winners always match highest successful bidders requires a restriction between any pair of `placeBid'` and `closeAuction'` operations. A simple scheme would be forcing instances of either operation to pay the same coordination cost. However, since `placeBid'` is likely to be more prevalent than `closeAuction'`, reducing the latency for `placeBid'` and penalizing `closeAuction'` is likely to lead to better performance.

To address this, we propose a coordination service called Olisipo offering a range of coordination policies, each of which presents a trade-off between the cost of each operation and the overall cost. This service allows us to use runtime information about the relative frequency of operations to select an efficient coordination mechanism for a given restriction.

5.1 Coordination protocols

Olisipo supports two built-in protocols, namely `symmetric (Sym)` and `asymmetric (Asym)`, but can be extended with customized coordination policies, which need to be compatible with our interface (Fig. 3). The difference between the two protocols is that, in the case of `Sym`, given a restriction $r(u,v)$ between two operations u and v, the protocol requires both u and v to coordinate with each other for establishing an order between them, whereas the `Asym` protocol allows one of them to proceed by default, while requiring the other to obtain permission before proceeding.

Sym. This protocol requires to set up a logically centralized counter service, which maintains, for each restriction $r(u,v)$, two counters c_u and c_v. Each one represents the total number of operations of the corresponding type that have been accepted by the underlying system. Additionally, every replica at different data centers maintains a local copy of these counters, representing the number of operations of each type that have been executed by that replica. Initially, all local copies, as well as the global counters, have all values set to zero. Whenever an operation is received by a replica, that replica contacts the counter service to increase the corresponding centralized counter and get a fresh copy of the counter maintained for both types of operations. Upon receiving the reply from the counter service, that replica compares the received values with its local copy. If they are the same, then the replica can execute the operation without waiting. Otherwise, the local execution can only take place when all missing operations have been locally replicated. To make the counter service fault tolerant, we leverage a Paxos-like state machine replication library (BFT-SMART [18]) to replicate counters across geo-locations.

Asym. Unlike the above centralized solution, the asymmetric protocol implements distributed barrier in a decentralized manner. Assume, for instance, that u is the barrier. In this case, whenever a replica r receives an operation u it would have to enter the barrier, and contact all other replicas to request participation. This requires all replicas in the system to stop processing operations of type v and enter the barrier. After receiving an acknowledgment of the barrier entrance from all replicas, r can execute the operation, and then notify all replicas that it has left the barrier (while at the same time propagating

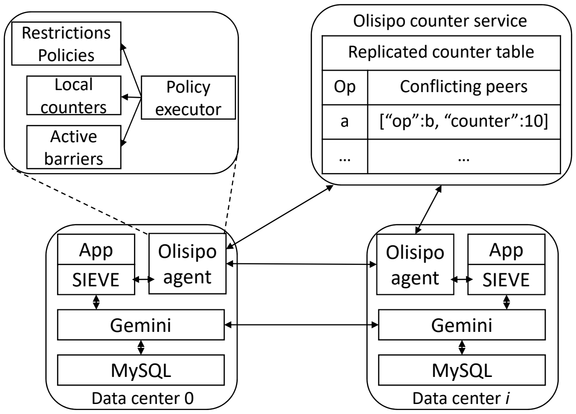


Figure 4: Olisipo architecture

the effects of the operation u it has just executed). Such a coordination strategy might incur a high overhead; however, this is beneficial when one of the two operations in the restriction is rarely submitted to the system.

5.2 Implementation details

As depicted in Fig. 4, the Olisipo architecture consists of a counter service replicated across data centers and a local agent deployed in each data center. The counter service is required only for the `Sym` protocol, whereas the local agent enforces the coordination that is needed by both protocols. To this end, every agent also stores some meta data required for different protocols: for the `Sym` protocol, it maintains a local copy of the replicated counter service, which is used for learning if the local counters lag behind the global counters, which means the corresponding data centers have to wait until all missing operations have been locally incorporated. For the `Asym` protocol, every agent maintains a list of active barriers, which are used for locally deciding if relevant operations blocked on such barriers can proceed. Note that these protocols are not optimized for performance, but nonetheless suffice to demonstrate the benefits of using PoR consistency and enforcing it in a way that takes into account frequency of different operation types.

We implemented Olisipo using Java (2.8k lines of code), linked with BFT-SMART [39] for replicating the centralized counter service and MySQL as the backend storage. We integrated Olisipo with our prior prototypes for Gemini [5] and SIEVE [6], so that Gemini serves as the underlying causally consistent replication tier while SIEVE is used to produce commutative shadow operations at runtime. The code of Olisipo is available at [11].

Workflow. User requests are directed to an application server running at the local data center, which executes the corresponding generator operation. The result is a commutative shadow operation, which is then forwarded to the local Olisipo agent for placing coordination if needed before committing; if the coordination allows for that serialization (which is determined according to the specific protocol for enforcing it) then the shadow operation is sent to Gemini for replicating it across all data centers; otherwise the generator operation must be retried in a new serial order.

6 Evaluation

In our experimental evaluation, we first try to understand if the methodology for inferring restrictions presented in Sec. 4 is effective when applied to real world applications, i.e., it finds a minimal set of restrictions. Finally, we try to assess the impact on latency and system throughput introduced by three factors: adopting PoR consistent replication, using different protocols, and adding more restrictions.

6.1 Case study

Next, we report our experience on discovering restrictions in RUBiS. RUBiS is a fairly simple auction-like benchmark. The original benchmark we used as a starting point contained only 16 transactions and did not include an operation to declare the winner of an auction, so we added a close auction functionality. In the future, we intend to explore other benchmarks with more complex OLTP or OLAP queries.

State convergence. Given that we deploy RUBiS with SIEVE, all shadow operations generated at runtime commute w.r.t. each other by construction, and there is no need modify the application nor restrict any pair of shadow operations for state convergence purposes. All that is required in SIEVE is to specify the desired conflict resolving semantics by choosing from a set of built-in solutions [29].

Invariant preservation. We manually perform the procedure of identifying restrictions to make a geo-replicated RUBiS deployment invariant preserving, as previously presented in Section 4.3. (We leave the automation of this step as future work.) In particular, we determined four invariants of RUBiS, namely (a) identifiers assigned by the system are unique; (b) nicknames chosen by users are unique; (c) item stock must be non-negative; and (d) the auction winner must be associated with the highest bid across all accepted bids. We continued by manually determining the weakest preconditions and postconditions of all RUBiS shadow operations. Those conditions are summarized in Tab. 1 and used by the `I-conflict set` analysis (Alg. 2). With regard to the first invariant, since we take advantage of the coordination-free unique identifier generation method offered by SIEVE, no `I-conflict sets` were found for violating it. In turn, for the remaining three invariants, we identified the following `I-conflict` sets:

- $\{registerUser', registerUser'\}$. Invariant (b) would be violated if the two operations proposed the same *nickname* and were submitted to different sites simultaneously;

$placeBid'$ $(itId, cId, bid)$	wp	$\exists u \in item\_table.\ u.id = itId \wedge u.status = open$	valid auction
	post	$bidTable = bidTable \cup \{< itId, cId, bid >\}$	new bid placed
$closeAuction'$ $(itId, wId)$	wp	$\exists w \in bidTable.\ w.cId = wId \wedge \forall v \in bidTable \setminus \{w\}.\ w.bid > v.bid$	highest accepted bid
	post	$winnerTable = winnerTable \cup \{< itId, wId >\}$	winner declared
$registerUser'$ $(uId, username)$	wp	$\forall u \in user\_table.\ u.name <> username$	username not seen before
	post	$user\_table = user\_table \cup \{< uId, username >\}$	new user added
$storeBuyNow'$ $(itId, delta)$	wp	$\exists u \in item\_table.\ u.id = itId \wedge u.stock >= delta$	enough stock left
	post	$u.stock -= delta$	delta applied

Table 1: Weakest preconditions and postconditions of selected shadow operations of RUBiS

RedBlue consistency	PoR consistency
$r(registerUser', registerUser')$	$r(registerUser', registerUser')$
$r(storeBuyNow', storeBuyNow')$	$r(storeBuyNow', storeBuyNow')$
$r(placeBid', placeBid')$	$r(placeBid', closeAuction')$
$r(closeAuction', closeAuction')$	
$r(placeBid', closeAuction')$	
$r(registerUser', storeBuyNow')$	
$r(registerUser', placeBid')$	
$r(registerUser', closeAuction')$	
$r(storeBuyNow', placeBid')$	
$r(storeBuyNow', closeAuction')$	

Table 2: Restrictions required when replicating the extended RUBiS under RedBlue or PoR consistency

- $\{storeBuyNow', storeBuyNow'\}$. Invariant (c) would be violated if both operations simultaneously subtracted some number of items from *stock*, and the sum of the purchases exceeded the previous *stock* value;
- $\{placeBid', closeAuction'\}$. Invariant (d) would be violated if both operations were submitted at the same time to different sites, and $placeBid'$ carried a higher bid than all accepted bids.

Each `I-conflict set` above covers a class of violating executions of the respective invariant. To eliminate the corresponding violations, we added three restrictions, namely $r(registerUser', registerUser')$, $r(storeBuyNow', storeBuyNow')$ and $r(placeBid', closeAuction')$. In Tab.2 we compare to the PoR consistency solution with using RedBlue consistency. The latter solution would require more restrictions, since the definition states that all non-invariant safe shadow operations must be strongly consistent, i.e., the four shadow operations presented in the above list must be restricted in a pairwise fashion.

We assign the `Sym` protocol to coordinate shadow operations confined by all these restrictions except $r(placeBid', closeAuction')$. This is because $placeBid'$ is significantly more prevalent than $closeAuction'$ in RUBiS, e.g., in a bidding mix workload, the ratio of the number of $closeAuction'$ to the number of $placeBid'$ is only 2.7%. Therefore, we assign the `Asym` protocol to coordinate this restriction and additionally make $closeAuction'$ act as the barrier.

6.2 Experimental setup

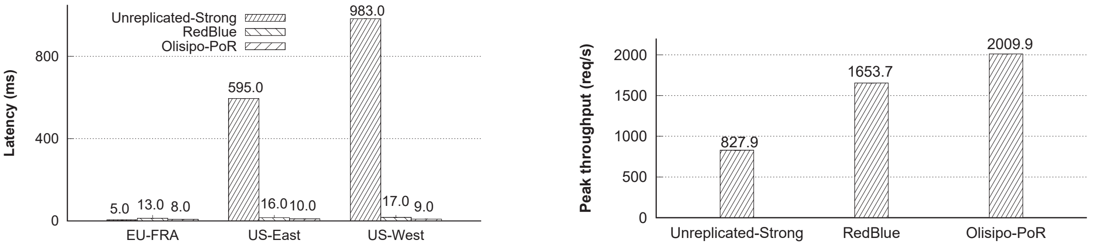
We run experiments on Amazon EC2 [1] using `m4.2xlarge` virtual machine instances located in three sites: US Virginia (US-East), US California (US-West) and EU Frankfurt (EU-FRA). Table 3 shows the average round trip latency and observed bandwidth between every pair of sites. Each VM has 8 virtual cores and 32GB of RAM. VMs run Debian 8 (Jessie) 64 bit, MySQL 5.5.18, Tomcat 6.0.35, and OpenJDK 8 software.

	US-East	US-West	EU-FRA
US-East	0.299 ms 1052.0 Mbps	71.200ms 47.4 Mbps	88.742 ms 29.6 Mbps
US-West	66.365 ms 47.4 Mbps	0.238 ms 1050.7 Mbps	162.156 ms 17.4 Mbps
EU-FRA	88.168 ms 36.2 Mbps	162.163 ms 20.1 Mbps	0.226 ms 1052.0 Mbps

Table 3: Average round trip latency and bandwidth between Amazon data centers

Configuration and workloads. Unless stated otherwise, in all experiments, we deploy the BFT-SMART library under the crash-fault-tolerance model (CFT) with 3 replicas across three sites, and assign the replica at EU-FRA to act as the leader of the consensus protocol. We replicate RUBiS under PoR consistency across three sites using the previously mentioned combination of Olisipo, SIEVE, and Gemini. As additional baselines, we run an unreplicated strongly consistent RUBiS in the EU-FRA site, and a 3 site RedBlue consistency deployment, in which we replicate RUBiS via the PoR consistency framework but with the set of restrictions required by RedBlue consistency (shown in Tab.2). We refer to these three setups as "*Olisipo-PoR*", "*Unreplicated-Strong*", and "*RedBlue*", respectively. For all experiments, emulated clients are equally distributed across three sites and connect to their closest data center according to physical proximity.

We choose to run the bidding mix workload of RUBiS, where 15% of user interactions are updates. To allow the client emulator to issue the newly introduced `closeAuction` requests, we have to slightly change the transition table in the original RUBiS code by assigning a positive probability value for this request. The new transition table can be found here [10]. For all experiments we vary the workload by increasing the number of concurrent client threads in every client emulator, and disable the `thinking time` option so that there is no waiting time between two contiguous requests from the same client thread. We populate the data set via the following parameters: the RUBiS database contains 33,000 items for sale, 1 million users, and 500,000 old items.



(a) Overall average latency

(b) Peak throughput

Figure 5: Performance comparison between three system configurations

6.3 Results

6.3.1 Average user observed latency

The main advantage of adopting PoR consistent replication with Olisipo is to reduce user-perceived latency. To assess this improvement, we start by analyzing the average latency for users at each data center. In this set of experiments, each user issues a single request at a time in a closed loop.

As shown in Fig.5(a), all users except those in EU-FRA observe a lower latency in the *Olisipo-PoR* and *RedBlue* configurations, compared to the users from the same locations in the *Unreplicated-Strong* configurations. This improvement is because, under both PoR and RedBlue consistency, most requests are handled locally within a data center, whereas in the unreplicated setting, requests from users at the two US data centers have to communicate with EU-FRA, which incurs an expensive inter-datacenter communication. At a more detailed level, in the *Unreplicated-Strong* experiment, the raw latency values perceived by users at both US-East and US-West are higher than the round-trip time from the user to the server site (EU-FRA) because processing each request involves sending one or more images to the user.

Compared to *RedBlue*, *Olisipo-PoR* improves the average latency for users at the three sites by 38.5%, 37.5% and 47.1%, respectively. We further observed that users at EU-FRA in the replicated experiments experience a higher latency than users from the same region accessing an unreplicated RUBiS. This is due to the additional work required for incorporating remote shadow operations into the local causal serialization and placing coordination when needed for serializing conflicting requests. Note that although the user observed latency for *Olisipo-PoR* at EU-FRA is almost twice as large as the latency of the unreplicated setup, the absolute number (9 ms) is reasonably low.

6.3.2 Peak throughput

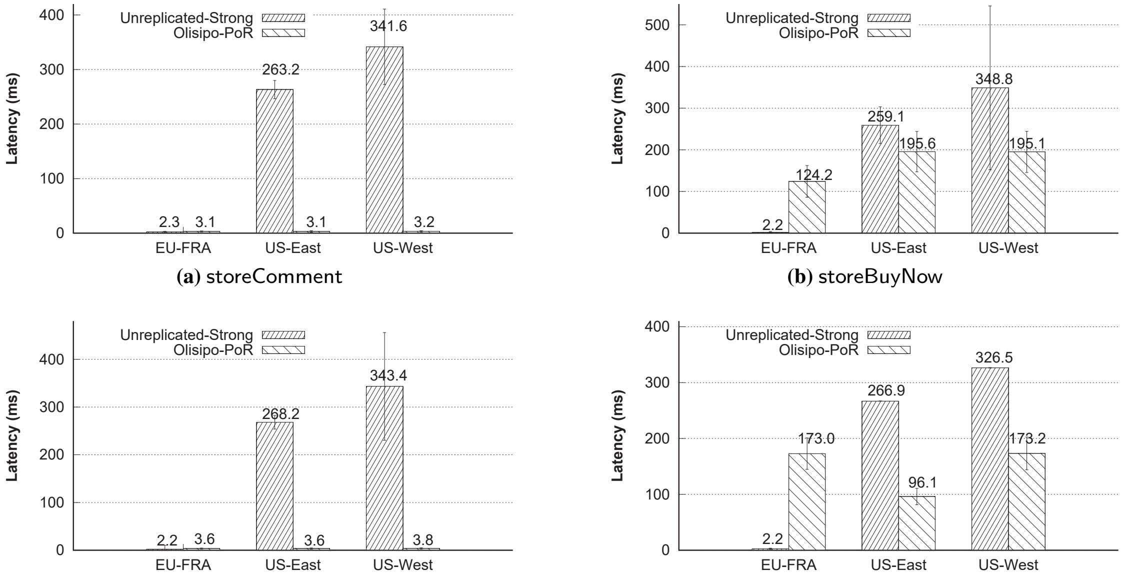
We now focus on the improvement in scalability with the client load achieved by PoR consistency. Fig.5(b) shows the peak throughput achieved by the three configurations, which is measured when the corresponding system is saturated. The improvement of the *Olisipo-PoR* deployment is $1.43X$ when compared to the *Unreplicated-Strong* setup. This increase in throughput is because PoR consistency offers fine-grained consistency so that only a minority of requests need to pay the coordination cost, while the remaining can be processed locally. Compared to a RedBlue consistent RUBiS, the PoR consistent version increases peak throughput by 21.5%, since PoR consistency avoids the cost for coordinating several restrictions required by RedBlue consistency (shown in Tab.2).

6.3.3 Per request latency

Next, we evaluate the per request latency of RUBiS requests. For this round of experiments, each site runs a single user issuing a request at a time.

Latency of non-conflicting requests. Among all RUBiS non-conflicting requests, we chose one representative request called `storeComment`, which places a comment on a user profile, as the illustrating example. Fig.6(a) shows that PoR consistent RUBiS makes users across the three sites observe evenly low latency, and the speedup in the user observed latency for the remote users located at US-East and US-West is 84.9x and 106.8x, respectively, compared to the *unreplicated* strongly consistent deployment. These performance gains happen because, under PoR consistency, the `storeComment` request requires no coordination and can be processed locally. In contrast, in the *unreplicated* experiment, users at the two US sites have to contact the server at EU-FRA and thus perceive a higher latency. We also notice that users from EU-FRA in both experiments have almost identical latency, which is different from the results in Fig.5(a), since the cost of generating and applying the shadow operations of the `storeComment` request is modest.

Latency of conflicting requests. Next, we shift our attention from non-conflicting requests to conflicting ones. As introduced before, Olisipo uses two different protocols (`Sym` and `Asym`) to coordinate conflicting requests. We start by analyzing the latency of requests handled by the `Sym` protocol. The illustrative example we selected is `storeBuyNow`, which produces self-conflicting shadow operations. As shown in Fig.6(b), the user observed latency of the `storeBuyNow` request at all three sites is significantly higher than the latency of `storeComment` (shown in Fig.6(a)), which is a non-conflicting request. This is because most of the lifecycle of these requests was spent asking permission to the centralized counter



(a) storeComment

(b) storeBuyNow

(c) placeBid

(d) closeAuction

Figure 6: Average latency bar graph of four requests for users at three sites. `storeComment` produces non-conflicting shadow operations, while the ones of `storeBuyNow` conflict w.r.t themselves and are regulated by the `Sym` protocol. `placeBid` and `closeAuction` produce two conflicting shadow operations regulated by the `Asym` protocol.

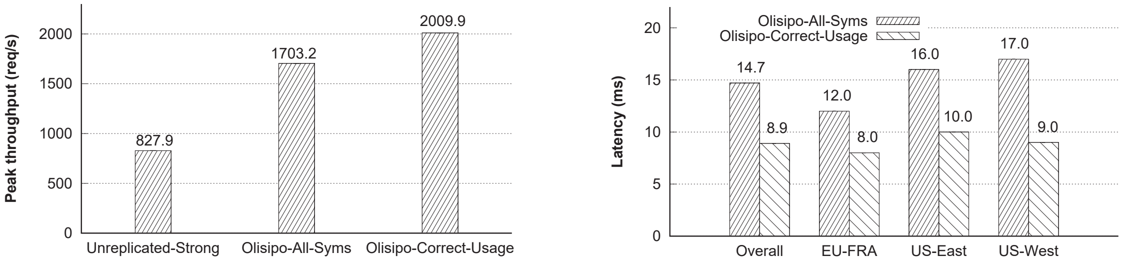
service, which consists of 3 replicas spanning three sites and executing a Paxos-like consensus protocol. Additionally, user observed latency at EU-FRA is lower than the remaining two sites, since the leader of the consensus protocol is co-located with EU-FRA users.

We continue by analyzing the average latency of requests that are coordinated by the `Asym` protocol. Unlike the `Sym` protocol, any pair of operations confined in a restriction will be treated differently by the `Asym` protocol, since one acts as a distributed barrier and the other proceeds if no active barriers are running. In Sect. 6.1, we assigned the `Asym` protocol to regulate the $r(placeBid', closeAuction')$ restriction, while selecting the less frequent shadow operation $closeAuction'$ as a barrier. As shown in Fig.6(c), the average latency measured for the *placeBid* request, which produces $placeBid'$, is very similar to the results obtained for non-conflicting requests shown in Fig.6(a). This is because the ratio of *closeAuction* to *placeBid* is very low and most of the time the *placeBid* request commits immediately without waiting for joining or leaving barriers.

Next, we consider the barrier request *closeAuction* handled by the `Asym` protocol. As expected, Fig.6(d) shows that, compared to *placeBid*, the average latency of *closeAuction* is noticeably higher due to the coordination across sites, through which this request forces all sites not to process incoming *placeBid* requests and collects results of all relevant completed *placeBid* requests. We also notice that users issuing *closeAuction* observed a latency that is slightly higher than the maximum RTT between their primary site and the remaining sites. For example, as shown in Tab.3, the maximum RTT for US-East users to the other two sites is 88.7 ms, while the average latency of *closeAuction* observed by the same group of users is 96.1 ms.

6.3.4 Impact of different protocols

The purpose of offering different coordination protocols is to improve runtime performance by taking into account the workload characteristics. To validate this, we first deploy an experiment denoted by `Olisipo-Correct-Usage`, in which we take into account the runtime information that $closeAuction'$ occurs sparsely and assign the `Asym` protocol to regulate the restriction $r(placeBid', closeAuction')$. We then deploy another experiment denoted by `Olisipo-All-Syms`, in which the restriction $r(placeBid', closeAuction')$ is handled by the `Sym` protocol. Fig. 7 summarizes the comparison of peak throughput and average latency among three experiments, namely `Unreplicated-Strong`, `Olisipo-All-Syms` and `Olisipo-Correct-Usage`. The `Olisipo-All-Syms` setup improves the peak throughput of the unreplicated RUBiS system by 105.7%, because of the coordination-free execution of non-conflicting requests. However, compared to `Olisipo-Correct-Usage`, the performance of `Olisipo-All-Syms` degrades in two dimensions, namely a 15.3% decrease in peak throughput and a 65.2%, 50.0%, 60.0%, 88.9% increase in request latency for all, EU-FRA, US-East, US-West users, respectively. The reason for this performance loss is as follows: every `placeBid'` shadow operation in `Olisipo-All-Syms` requires a communication step between its



(a) Peak throughput

(b) Overall average latency

Figure 7: Peak throughput and overall average latency bar graphs of systems using different protocols.

primary site and the centralized counter service for being coordinated, while most of time `placeBid'` shadow operations in `Olisipo-Correct-Usage` work as non-conflicting requests provided that `closeAuction` requests sparsely arrive in the system.

7 Related work

In the past decades, many consistency proposals focused on reducing coordination among concurrent operations to improve scalability in geo-replicated systems [25, 40, 30, 29, 14, 15, 42, 12, 4, 32]. However, they only allow the programmer to choose from a limited number of consistency levels that they support, such as strong, causal or eventual consistency. Unlike these approaches, PoR consistency offers a fine-grained tunable trade-off between performance and consistency using the visibility restrictions between pairs of operations to express consistency semantics. Some of these proposals for consistency models with reduced coordination also analyzed or even enforced conditions for ensuring state convergence despite the lack of coordination [14, 25, 40, 15, 32, 40]. In addition to state convergence, our solution also analyzes invariant preservation.

In the space of consistency proposals that looked into how to enforce application-specific invariants, Bailis et al. [16] proposed I-confluence, which avoids coordination by determining if a set of transactions are I-confluent, i.e., if the integrity constraints might be violated when they are executing without coordination. Indigo [17] defines consistency as a set of invariants that must hold at any time, and presents a set of mechanisms to enforce these invariants efficiently on top of eventual consistency. Similar to Indigo, warranties [31] map consistency requirements to a set of assertions that must hold in a given period of time, but it needs to periodically invalidate assertions when updates arrive. Roy et al. additionally propose a program analysis against transaction code for producing warranties [35]. In contrast, PoR consistency takes an alternative approach by modeling consistency as restrictions over operations.

A few proposals map consistency semantics to the ordering constraints defined over pairs of operations. Generic Broadcast defines conflict relations between messages for fast message delivery, which are analogous to visibility restrictions used in our solution [34]. However, they do not analyze how to determine the conditions for ensuring invariant preservation. The recent work of Gotsman et al. [24] encodes the concept of a conflict relation into a proof system, which allows for analyzing if consistency choices expressed as conflict relations is sufficient for enforcing application invariants. In comparison, our work makes three contributions. First, our methods allow to find a minimal set of restrictions to be used. Second, we propose a set of coordination methods that adapt to the workloads in order to be more efficient. Third, we present the design and implementation of a complete system that offers PoR consistency.

Some variants of Paxos [26] have explored operation semantics to relax the need to process all operations in the same sequential order. Generalized Paxos allows replicas to execute commutative operations in different orders [27]. EPaxos uses dependencies between pairs of operations to order concurrent conflicting requests [33]. Our work differs from these Paxos variants in that we develop an analysis to extract pairs of conflicting operations by considering the impact of concurrent executions on achieving state convergence and invariant preservation. Furthermore, unlike these protocols, in our work, operations that are not confined by conflicting relations can be first accepted in a single replica and later asynchronously replicated to other replicas.

Finally, our own previous workshop paper described the motivation and a high-level overview of a solution to this problem [28].

8 Conclusion

In this paper, we proposed a technique for achieving convergence and invariant-preservation in geo-replicated systems with a minimal amount of coordination. This combines a new generic consistency model called PoR consistency, an analysis for determining a minimal set of restrictions, and a coordination service called Olisipo for efficiently serializing pairs of operations. Our evaluation of running RUBiS with different setups shows that the joint work of PoR consistency and Olisipo significantly improves the performance of geo-replicated systems.

Acknowledgments

We sincerely thank Robbert van Renesse, Jiawei Wang, Xinyu Feng, and the anonymous reviewers for their insightful comments and suggestions. The research of C. Li is supported by the Fundamental Research Funds for the Central Universities (Grant no. WK2150110011). This work is supported by the Portuguese Funda cão para a Ciência e a Tecnologia through projects UID/CEC/50021/2013 and UID/CEC/04516/2013 and by the EU H2020 LightKone project (732505).

References

[1] Amazon Elastic Compute Cloud (EC2). https://aws.amazon.com/ec2/. [Online; accessed Jan-2018].

[2] Amazon S3 Introduces Cross-Region Replication. https://aws.amazon.com/about-aws/whats-new/2015/03/amazon-s3-introduces-cross-region-replication/. [Online; accessed Jan-2018].

[3] Amazon Web Services (AWS) - Cloud Computing Services. http://aws.amazon.com/. [Online; accessed Jan-2018].

[4] Balancing Strong and Eventual Consistency with Google Cloud Datastore. https://cloud.google.com/datastore/docs/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore/. [Online; accessed Jan-2018].

[5] Gemini Code Repository. https://github.com/pandaworrior/RedBlue_consistency. [Online; accessed Jan-2018].

[6] SIEVE Code Repository. https://github.com/pandaworrior/SIEVE. [Online; accessed Jan-2018].

[7] Extended version with proofs. https://github.com/mr-cheng-li/por_tr. [Online; accessed Feb-2018].

[8] Google Webpage. www.google.com. [Online; accessed Jan-2018].

[9] Microsoft US — Devices and Services. www.microsoft.com/. [Online; accessed Jan-2018].

[10] Modified RUBiS Transition Table. https://github.com/pandaworrior/VascoRepo/blob/master/vasco/config/vasco_transitions_3.xls. [Online; accessed Jan-2018].

[11] Olisipo code repository. https://github.com/pandaworrior/VascoRepo. [Online; accessed Jan-2018].

[12] Welcome to Azure Cosmos DB. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction. [Online; accessed Jan-2018].

[13] Welcome to Facebook - Log In, Sign Up or Learn More. https://www.facebook.com/. [Online; accessed Jan-2018].

[14] Alvaro, P., Conway, N., Hellerstein, J. M., and Maier, D. Blazes: Coordination Analysis for Distributed Programs. In *Proceedings of the IEEE 30th International Conference on Data Engineering* (2014), ICDE'14.

[15] Alvaro, P., Conway, N., Hellerstein, J. M., and Marczak, W. R. Consistency Analysis in Bloom: a CALM and Collected Approach. In *Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research* (2011), CIDR'11.

[16] Bailis, P., Fekete, A., Franklin, M. J., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Coordination Avoidance in Database Systems. *Proc. VLDB Endow. 8*, 3 (Nov. 2014), 185–196.

[17] Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N., Najafzadeh, M., and Shapiro, M. Putting Consistency Back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 6:1–6:16.

[18] Bessani, A., Sousa, J. a., and Alchieri, E. E. P. State Machine Replication for the Masses with BFT-SMART. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2014), DSN '14, IEEE Computer Society, pp. 355–362.

[19] Burckhardt, S., Gotsman, A., and Yang, H. Understanding Eventual Consistency. Tech. Rep. MSR-TR-2013-39, March 2013.

[20] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., Haq, M. F. u., Haq, M. I. u., Bhardwaj, D., Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K., and Rigas, L. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 143–157.

[21] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[22] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[23] Dijkstra, E. W. *A Discipline of Programming*, 1st ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[24] Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., and Shapiro, M. 'Cause I'm strong enough: reasoning about consistency choices in distributed systems. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2015), POPL'15, ACM.

[25] Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. Providing High Availability Using Lazy Replication. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992), 360–391.

[26] Lamport, L. The Part-time Parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[27] Lamport, L. Generalized Consensus and Paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research, 2005.

[28] Li, C., Leitão, J. a., Clement, A., Preguiça, N., and Rodrigues, R. Minimizing Coordination in Replicated Systems. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data* (New York, NY, USA, 2015), PaPoC '15, ACM, pp. 8:1–8:4.

[29] Li, C., Leitão, J. a., Clement, A., Preguiça, N., Rodrigues, R., and Vafeiadis, V. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the*

2014 USENIX Conference on USENIX Annual Technical Conference (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 281–292.

[30] LI, C., PORTO, D., CLEMENT, A., GEHRKE, J., PREGUIÇA, N., AND RODRIGUES, R. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 265–278.

[31] LIU, J., MAGRINO, T., ARDEN, O., GEORGE, M. D., AND MYERS, A. C. Warranties for Faster Strong Consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 503–517.

[32] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 401–416.

[33] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 358–372.

[34] PEDONE, F., AND SCHIPER, A. Generic Broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing* (1999), DISC '99.

[35] ROY, S., KOT, L., BENDER, G., DING, B., HOJJAT, H., KOCH, C., FOSTER, N., AND GEHRKE, J. The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1311–1326.

[36] SCHURMAN, E., AND BRUTLAG, J. Performance Related Changes and their User Impact. http://slideplayer.com/slide/1402419/, 2009. Presented at *Velocity Web Performance and Operations Conference*. [Online; accessed Jan-2018].

[37] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Tech. Rep. 7506, INRIA, Jan. 2011.

[38] SHARMA, Y., AJOUX, P., ANG, P., CALLIES, D., CHOUDHARY, A., DEMAILLY, L., FERSCH, T., GUZ, L. A., KOTULSKI, A., KULKARNI, S., KUMAR, S., LI, H., LI, J., MAKEEV, E., PRAKASAM, K., VAN RENESSE, R., ROY, S., SETH, P., SONG, Y. J., VEERARAGHAVAN, K., WESTER, B., AND XIE, P. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 351–366.

[39] SOUSA, J., ALCHIERI, E., AND BESSANI, A. BFT-SMART Code Repository. https://github.com/bft-smart/library. [Online; accessed Jan-2018].

[40] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional Storage for Geo-replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 385–400.

[41] VOGELS, W. Eventually Consistent. *Commun. ACM 52*, 1 (Jan. 2009), 40–44.

[42] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 263–278.

Log-Free Concurrent Data Structures*

Tudor David† — *IBM Research, Zurich*
Aleksandar Dragojević — *MSR Cambridge*
Rachid Guerraoui — *EPFL*
Igor Zablotchi — *EPFL*

Abstract

Non-volatile RAM (NVRAM) makes it possible for data structures to tolerate transient failures, assuming however that programmers have designed these structures such that their consistency is preserved upon recovery. Previous approaches are typically transactional and inherently make heavy use of logging, resulting in implementations that are significantly slower than their DRAM counterparts. In this paper, we introduce a set of techniques aimed at lock-free data structures that, in the large majority of cases, remove the need for logging (and costly durable store instructions) both in the data structure algorithm and in the associated memory management scheme. Together, these generic techniques enable us to design what we call *log-free concurrent data structures*, which, as we illustrate on linked lists, hash tables, skip lists, and BSTs, can provide several-fold performance improvements over previous transaction-based implementations, with overheads of the order of milliseconds for recovery after a failure. We also highlight how our techniques can be integrated into practical systems, by presenting a durable version of Memcached that maintains the performance of its volatile counterpart.

1 Introduction

Fast, non-volatile memory technologies have been intensively studied over the past years, with various alternatives such as Memristors [53], Phase Change Memory [31, 49], and 3D XPoint [39] being proposed. Nevertheless, these technologies are only now starting to become commercially available. Referred to as *non-volatile RAM (NVRAM)*, they promise byte-addressability and latencies that are comparable to DRAM, yet also non-volatility and higher density than DRAM.

From a programmer's perspective, NVRAM can be read and written using *load* and *store* instructions, identically to DRAM. However, a significant fraction of software needs to be redesigned to work with NVRAM. Unlike DRAM on the one hand, in order to take advantage of NVRAM's non-volatility, the stored data needs to be in a state that allows the resumption of execution after a transient failure (e.g., a power failure). Unlike block-based durable storage on the other hand, the granularity at which data is read and written is much finer, and the latencies much smaller. Thus, strategies that might have yielded the best performance in case of block-based storage might not be appropriate for NVRAM.

In this paper, we focus on adapting to NVRAM the design and implementation of an essential component of modern software systems: concurrent data structures [10, 13, 38, 40, 41, 43]. Ideally, in the NVRAM environment, one would like concurrent data structures that (a) can be recovered in case of a transient failure with states that reflect all completed operations up to the failure, yet (b) whose performance and scalability resemble those of their counterparts designed for DRAM.

This task is challenging because neither data stored in registers, nor in caches, is durable in the face of transient failures. Moreover, by default, the program does not control the order in which cache lines are evicted and written to NVRAM. In order to enforce ordering, specific (and expensive) instructions, which we refer to as *sync* operations, must be used to ensure that a store is written through to NVRAM at the desired point.

Previous approaches [2, 4, 6, 20, 25, 28, 30, 33, 37, 56] to implementing data structures for NVRAM rely mainly on transactions (either explicitly, or implicitly derived from critical sections). With transactions, some form of *logging* is necessary to cope with the possibility of a failure in the middle of a transaction. This log needs to be reliably written before the transaction is executed. This entails waiting for stores to be written to NVRAM before proceeding, which is particularly expensive: whereas when using DRAM, one would at most wait for data to be written to the L1 cache, now one has to wait for data to be written all the way to NVRAM. Logging is thus a major source of expensive sync operations.

We propose three techniques aimed at lock-free data structures that remove logging entirely from data structure operations and dramatically reduce logging in the memory reclamation scheme. We focus on lock-free algorithms, because they always keep the data structure in a consistent state and thus do not inherently require logging in order to maintain consistency across restarts.

The techniques we propose are: (1) the *link-and-persist* technique, which allows atomically changing and persisting a link in a data structure, (2) the *link cache*, which allows persisting entire batches of modified links, thus reducing the number of *sync* operations and (3) *NV-epochs*, a coarse-grained epoch-based memory reclamation scheme. We briefly describe these techniques below.

Link-and-persist applies the pointer marking technique [15] from concurrent programming to ensure atomicity in the face of crashes and recoveries. With link-and-persist, when a data structure link is modified, a mark is

[*] Supported in part by ERC Grant 339539 (AOC).

[†] Work done while the author was at EPFL and MSR Cambridge.

added to the link to signify that the value of the link might not be durable yet. The link can then be persisted, and the mark removed, by the modifying operation or by any other operation (helping).

The *link cache* is an extremely fast, best-effort concurrent hash table stored in volatile memory, which contains data structure links that have not yet been durably written. When modifying the data structure, instead of ensuring updated links are written to NVRAM, we add them to the link cache. Thus, we avoid writing them to NVRAM one at a time. When the durable write of one of them is necessary for correctness, we batch the write-backs of all links stored in our cache, which is significantly faster than waiting for writes to complete one at a time.

Finally, memory allocation and reclamation is also a central concern for concurrent data structures. When working with NVRAM, the traditional approach for avoiding persistent memory leaks or use-after-free problems is again some form of logging. To avoid this, we propose *NV-epochs*, a coarse-grained epoch-based memory reclamation scheme for durable and concurrent data structures. NV-epochs groups memory nodes into memory areas, and reliably and durably keeps track of the active (recently used) memory areas instead of individual allocations. This bookkeeping of active memory areas can be seen as the only form of logging in our approach. However, we make the observation that most of the time, allocation and reclamation exhibit locality[1]. Therefore, logging can be sidestepped entirely in this case, because the memory area an operation accesses will already be marked as active, and thus we do not have to wait for any additional store for memory leak prevention. When recovering after a failure, we simply need to traverse the memory areas that were active at the moment of the crash and detect which objects belonging to these areas are still linked in the data structures. This is significantly faster than generic mark-and-sweep garbage collection for instance [1].

Each of these three techniques is of independent interest, and can be applied individually while maintaining its associated benefits. Together, these techniques can be used to produce what we call by abuse of language *log-free durable concurrent data structures*, namely, durable concurrent data structures that, in the large majority of cases described above, require no logging whatsoever. As we show in the paper, these data structures provide up to an order of magnitude faster updates than a traditional log-based approach, both in single-threaded and in concurrent environments. Moreover, we achieve these benefits while maintaining low recovery times in case of restarts: even for gigabyte-sized structures, the time required to recover the structure is of only a few milliseconds. In terms of correctness, our implementations guarantee durable linearizability [26]. Briefly, all the operations completed before a crash are reflected after recovery.

We also highlight the practicality of our techniques by developing *NV-Memcached*, a durable version of Memcached [38] that is based on a lock-free, durable hash table. NV-Memcached performs similarly to the volatile memory version of Memcached.

Still, our approach is not a silver bullet. While it largely removes the cost of logging for all data structure sizes, this is especially beneficial for small and medium-sized data structures. Indeed, (1) these data structures exhibit high locality in memory allocation/deallocation, as we show in the paper, and (2) the relative cost of sync operations is higher for these data structures, as opposed to larger structures, where other costs, such as traversal, dominate.

To summarize, the contributions of this paper are:

1. *Link-and-persist*: a methodology for designing data structures with no logging in the main operations;
2. *Link cache*: a component that largely eliminates sync operations in durable data structures;
3. *NV-epochs*: a durable memory management scheme in which only a fraction of operations do any logging;
4. *NV-Memcached*, a durable version of Memcached based on our techniques;
5. A library of log-free durable data structures, as well as the link cache, NV-epochs, and NV-Memcached implementations, all available at `go.epfl.ch/nvram`.

The rest of the paper is organized as follows. In § 2 we recall relevant background. We describe our link-and-persist technique in § 3. We discuss our link cache in § 4, and memory management in § 5. We show experimental results in § 6 and discuss related work in § 7.

2 Background

Traditionally, storage has either been fast, but volatile (i.e., data is lost in case of a power failure), as is the case with DRAM, or non-volatile, but slow, as is the case with flash storage for instance. However, more recently, a new class of storage that promises low latency, byte-addressability, and non-volatility is becoming available. NVRAM latencies are expected to be somewhat larger than those of DRAM, with writes being more expensive than reads. Table 1 compares expected PCM and Memristor latencies [51, 54, 59] to those of DRAM and caches.

As highlighted in the introduction, one of the main difficulties when working with NVRAM stems from the fact that, by default, we do not control the order in which cache lines to which we have performed stores are evicted from the caches, and actually written to NVRAM. However, there are current and upcoming instructions [21, 23] on Intel processors[2], which allow us to ensure that a cache line is indeed written to memory. In this paper, we consider

[1] For small and medium sized data structures, as we show, this covers more than 99% of memory operations.

[2] In this work, we assume a TSO-like memory model; our work can be extended to more relaxed memory models as well.

	L1	L2	LLC	DRAM	PCM	Memristor
Read	2	6	15	50	50-70	100
Write	2	6	15	50	150	100

Table 1: *Caches, DRAM, and NVRAM (projected) latencies (ns).*

the *clwb* instruction, because it (a) writes-back a cache line without invalidating it—as opposed to *clflushopt*—and (b) it is only ordered with respect to fences (or to instructions that have an implied store fence, such as, for example, Compare-and-Swap)—as opposed to *clflush*. Property (b) is especially beneficial to performance, since it allows multiple cache-line write-backs to proceed in parallel [22]. We refer to one or more such instructions followed by a store fence as a *sync* operation.

A machine may fail at any point in time (e.g., due to a power failure), but can be expected to restart and resume normal operation (transient failure). We assume, as is commonly done in practice, that only the data stored in durable main memory is still available after a crash. The data that was in a processor's registers or in the write-back caches at the moment of the crash is not available after a restart. Nevertheless, our approach would be highly beneficial and remove the need for logging on an architecture that maintains enough residual energy to flush the register and the caches in case of a power failure as well.

Similar to related work [1], we assume that a region of NVRAM can be mapped to the same region of virtual memory across restarts. Alternatively, if this is not the case, we can update persistent pointers at recovery time.

In the context of concurrent software, it is important to define correctness conditions in the face of restarts. For this purpose, we use the concept of *durable linearizability* introduced by Izraelevitz et al. [26]. Essentially, a durably linearizable implementation guarantees that the state of the data structure after a restart reflects a consistent operation subhistory that includes all the completed operations at the moment of the crash.

3 The Link-and-Persist Technique

In this section, we present our *link-and-persist* technique for designing concurrent and durable data structures. We first argue that such data structures should be lock-free, then we detail the technique itself and how it can be used to obtain correct lock-free data structure implementations for NVRAM. We focus on implementations of linked lists, skip lists, hash tables, and search trees, which are commonly used in practice [10, 13, 38, 40, 41, 43]. Nevertheless, our techniques also apply to other data structures.

The Case for Lock-Free Algorithms. As noted by previous work [7, 26, 29, 46, 47], lock-free algorithms are a good fit for the NVRAM environment. This is because in lock-free algorithms, threads must ensure that the data structure is in a consistent state at all times, so that the failure of any number of threads does not prevent remaining threads from making progress. A beneficial consequence is that, when used with NVRAM, lock-free algorithms ensure that as long as threads' stores are persisted in the order in which they are issued (we show how this can be relaxed), regardless of when a crash occurs, upon a restart the data structure is in a consistent state that allows the execution to resume. Therefore, we remove the need for logging for the data structure itself, as opposed to transactional approaches which inherently require logging.

Our Technique. In linked data structures, a new node becomes visible when a link to it from an existing node is atomically inserted. Once this happens, other operations can see that the new node is present. Furthermore, in many algorithms, a node becomes logically deleted when a mark is atomically inserted on a link to signal deletion. After this, all operations enquiring about the state of this node will consider the node as no longer in the structure. A node becomes unreachable when the last link to it from another node in the data structure is atomically removed.

All these operations change the state of the node, and determine the return value of other operations which depend on the particular node. In the context of NVRAM, in order to ensure durable linearizability, it is therefore essential that all direct dependencies of an operation be durably written before the operation is performed. Otherwise, a scenario in which the user receives a return value, the system restarts, and the stored data no longer reflects the state observed by the user is possible.

In order to deal with this issue, the most straightforward approach is what we call the *link-and-persist* operation. Essentially, when performing a link update that changes the state of a node, the link is atomically updated normally, but contains a mark to signal that there is no guarantee its state is persisted. The updating operation then persists the newly modified link, and once the link is guaranteed to be persisted, it atomically removes the mark. If another operation whose result depends on the marked link occurs before the updating thread can persist it and remove the mark, the second operation will try to do these steps itself. This method involves no blocking, and introduces bounded overhead, thus being suitable for all concurrent algorithm classes, including lock-free and wait-free algorithms [17].

We illustrate our technique through the example of a lock-free linked list that uses the algorithm proposed by Tim Harris [15]. In the original (volatile) algorithm, in the case of inserts, once a node is properly allocated and initialized, we simply have to set the *next* pointer of its predecessor to point to it. In the case of a delete, we must first atomically flag the *next* pointer of the node to be deleted to signal logical deletion, after which the *next* pointer of its predecessor is set such that it bypasses the node to be deleted. Figure 1 shows the extra steps taken when inserting a new node using link-and-persist.

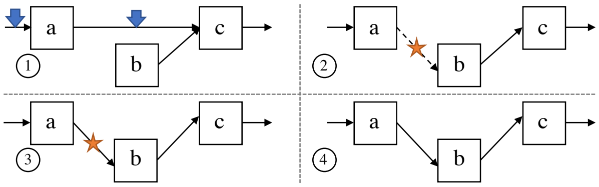


Figure 1: Stages of inserting a node in a linked list using link-and-persist. 1.) The new node (b) is created and its predecessor's (a) adjacent links (indicated by downward arrows) are persisted. 2.) The predecessor node's next pointer is atomically made to point to the new node. A mark (⋆) is added to the link to indicate it is not durable yet. 3.) The new node's incoming link is persisted. 4.) The mark is removed from the incoming link.

Durable Implementations. We have used link-and-persist to implement several durable data structures: linked lists, hash tables, skip lists, and BSTs. These structures model the set abstraction, and have methods to insert, remove, and search for elements identified through a unique key. We consider one implementation per data structure type, starting from the concurrent algorithm that has been shown to provide the best performance and scalability [8]. Our linked list is based on Harris' algorithm [15], the hash table uses one Harris linked list per bucket, the skip list uses the lock-free algorithm by Herlihy et al. [19], while the BST uses the algorithm proposed by Natarajan and Mittal [45]. Other algorithms and data structures can be similarly modified.

Correctness. Our data structures are linearizable, since we start from linearizable algorithms and add only flushes, which do not impact linearization points. We also ensure two additional properties [12]. First, each update operation ensures that its changes are durable before returning. Second, each operation O ensures that all operations O depends on (that involve some of the same nodes and were already linearized) are also durably linearized before O makes changes. Together, these two properties ensure durable linearizability, because they ensure that after a restart, the data structure reflects a consistent cut [26] of the history including all operations that completed before the crash and potentially some operations that were ongoing when the crash occurred.

The first property is easily ensured by durably writing any new edges or nodes introduced by an operation. Making sure an edge e is durably written just means checking if e is marked, and issuing write-backs only if e is not yet durable. The second property is achieved because operations ensure that (1) before an edge is modified, the edge is durably written and (2) incoming and outgoing edges (*adjacent edges*) of nodes involved in the operation are durable before proceeding.

We detail point (2) on a linked list (similar considerations apply for the other linked data structures). For a successful search, we make sure adjacent edges to the returned node are durably written before returning. For a failed search, we make sure the node is durably unreachable before returning (e.g., in the case where a node is marked but not yet durably unlinked). For the parse phase of a modify operation (insert or delete) [8], we take the same steps as for a search. For an insert, we also ensure that adjacent edges to the predecessor are durable before linking the new node. For a delete, we ensure that adjacent edges to the target node T and to T's predecessor are durable before unlinking the target node. In all cases, if an edge e has changed between the time e is read and the time we try to durably write e, then the operation that changed e made sure e was durable.

4 Limiting Write-Backs: the Link Cache

We now introduce our second technique, aimed at further reducing the number of *sync* operations in durable data structures.

4.1 Link Cache Overview

As discussed in § 2, batching multiple cache line write-backs is significantly faster than persisting them one at a time. Therefore, we propose the following scheme: when doing an update, do not immediately persist links, but place them in a fast, *volatile* cache (the *link cache*), and write back all the links in this cache when an operation that directly depends on one of them occurs. Of course, this means that clients which have inserted links into the link cache can only consider the operation completed once the link cache is flushed to NVRAM. The changes of a link and the insertion of a corresponding entry in the link cache must occur atomically (achievable in a non-blocking manner by using hardware transactional memory, or by marking the pointers to be inserted in the link cache while the operation is ongoing). If a restart happens, modified links currently in the link cache might be lost. However, this is not problematic: the fact that these link addresses were in the cache at the moment of restart means that no operation that directly depends on them completed, and thus its outcome may or may not be visible. We thus maintain the durable linearizability property. In addition, an atomic update of an ongoing operation not being durably recorded does not leave the data structure in an incorrect state after a restart. Where ordering of durable updates is necessary, we enforce it in the data structure algorithm (see § 3). The link cache is practical as long as inserting an entry in the cache is faster than waiting for a cache line to be written back to NVRAM.

4.2 Link Cache Implementation

Our main aims for the link cache are small memory footprint, non-blocking operation, and fast insertions. With these requirements in mind, we chose to make insertions in the cache best effort. The cache is only useful if it can improve the time updates spend waiting. Therefore, if an

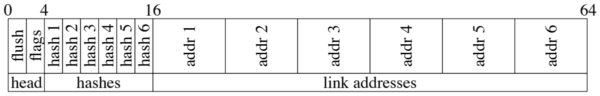


Figure 2: *A bucket in the link cache.*

update attempts to insert an entry in the cache, but does not succeed on the first try, it gives up and persists the modified link itself instead of waiting. Thus, link cache insertions have constant worst case performance.

Our hash table has a configurable (but fixed throughout the execution) number of buckets. Each bucket spans exactly one cache line, and can store up to 6 links. Links concerning a particular key map to one and only one bucket. Figure 2 details the contents of a bucket. The first two bytes are used to signal whether the bucket is currently being flushed. The next two bytes are used to store the current state of each of the entries in the bucket. An entry can be free, pending or busy. We next store the 6 keys associated with the links in the bucket. In order to be able to fit 6 entries in a single cache line, instead of storing the entire key, we only store a 2-byte hash for each of the keys. While this might result in false collisions, they are extremely unlikely. With 32 buckets, we have a hash space of size 2M. Even if false collisions do occur, this is not problematic: we would simply be triggering a flush of the links in the cache when this might not have been strictly necessary. The hashes therefore require 12 bytes in each bucket. The remaining 48 bytes in the cache line are used to store the addresses of the 6 links.

The interface of the link cache has three operations, which we discuss in the following.

Try Link and Add. If there is space in the link cache, this operation atomically modifies the link in the data structure and inserts an entry in the link cache. The operation first tries to reserve an entry in the link cache. To this effect, it tries to atomically change the state of an entry from free to pending. If no free entry exists, the link cache is being flushed, or the attempt to reserve an entry is not successful, the caller is notified that the operation did not succeed and that it should persist the link itself. Once an entry is reserved, we set the corresponding key and link address in the link cache. Next, we try to update the link in the data structure. We insert the new link, but use a bit to mark the fact that for now, this link has been neither persisted, nor has its addition to the link cache been marked as completed. If the link update fails, we set the state of the link cache entry to free and return failure to the caller. We next set the state of the entry in the link cache to busy (to mark the fact that we have added the key and link address, and that the link address in fact contains the value that we want to persist). Finally, we remove the mark from the link in the data structure.

The fact that this operation is best effort, and the fact that we do several atomic updates (link marking, transitioning between multiple states) just in order to be able to handle concurrent readers make this operation an ideal candidate for the use of hardware transactional memory (HTM). In fact, we first try to execute a fast-path HTM-based operation before reverting to the code presented above. In the HTM path, we do not need to insert a marked link into the data structure, and we can avoid going through the pending state in the link cache.

Flush. This operation writes all the finalized entries in a bucket to NVRAM. The operation first atomically increments the corresponding flushing counter to signal that it is in the process of flushing a bucket. The flush operation then issues write-backs for the link addresses in the busy entries in the bucket one by one (without waiting for the write-backs to complete) and sets the state of these entries to free. Next, the operation checks if any of the entries we have not written back have become busy (completed) in the meanwhile, and if yes, issues write-backs for them as well. This is repeated until no new busy entries appear. The thread then waits for the write-backs to complete by issuing a fence, decrements the flushing counter, and returns.

Scan. The scan operation is given a key and searches for any link pertaining to this key in the link cache. If such an entry is found in busy state (i.e., the insertion of the link was finalized), a flush is triggered. If an entry is found but is in pending state, the operation checks whether the new pointer has been inserted into the data structure. If this is the case, the current operation's linearization point should be after that of the operation currently inserting into the link cache, and therefore the current operation triggers a write-back of the new value of the link. Otherwise, the current operation's linearization point is before that of the update, and no further action needs to be taken. In order to guarantee durable linearizability, every data structure operation needs to call the scan method for its key, as well as for its predecessor in the structure in case of updates. However, this is as fast as reading two cache lines.

Illustration: Link Cache Effectiveness. We illustrate the effectiveness of the approach using the same linked list example employed in the previous section. The schedule of operations in our example, as well as the way the link cache is constructed are presented in Figure 3. We assume an initially empty link cache, and we only depict the effects of operations that change the state of the data structure or the link cache. Normally, updates would have to wait for one link to be persisted in the case of the insert operations, and two links in the case of the delete operation (one for marking and one for deletion). However, in this example, by using the link cache, we have replaced writing back 4 cache lines one at a time by a single batch of 3 cache line write-backs.

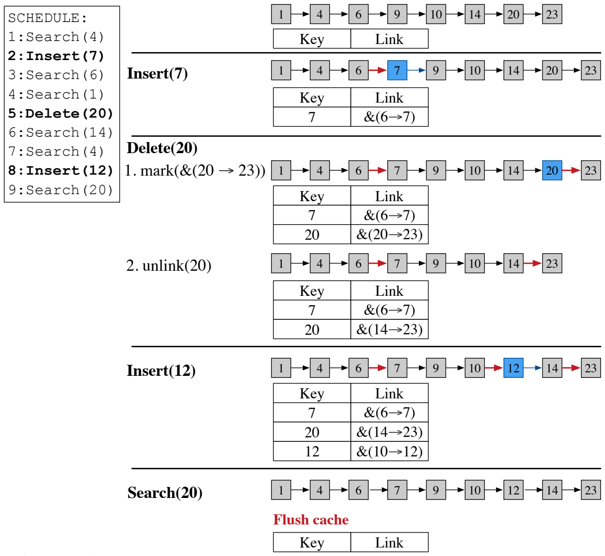


Figure 3: *Example of how the link cache is constructed.*

5 Memory Management with NV-Epochs

We now address another issue that is unavoidable whenever inserting or removing nodes in a linked concurrent data structure: memory management.

5.1 Overview

Two separate steps need to be performed both when inserting and removing a node: in case of an insertion, memory for the new node is first allocated and initialized, after which the node is linked into the data structure; in case of a deletion, the node is first unlinked from the data structure, and later, when we are sure no references to it exist, its memory is freed. If a restart occurs between these two main steps both in case of an insertion and a removal, a persistent memory leak occurs: we have allocated data that is not linked anywhere in our data structure.

The typical way of addressing the issue in the context of NVRAM is to use some form of logging: before allocating and linking, we log our intention, as we do before unlinking and freeing memory. Once the operation has (durably) completed, the log entry can be removed. However, this entails an extra write to NVRAM per update. In addition, this write needs to complete before we can proceed with the update, thus producing a non-negligible increase in the latency of updates.

In order to avoid waiting for the durable log to be written at each allocation or deallocation, we propose keeping track of coarser-grained active memory areas instead of keeping track of individual allocations/deallocations. Intuitively, when allocating, threads often reserve larger contiguous memory areas from which they serve user requests; thus, consecutive allocations tend to belong to the same memory area. In addition, memory reclamation schemes keep track of which objects have been unlinked, and periodically free those to which no references are held. This reclamation step is only run periodically for performance considerations (typically, when a certain number of unlinked objects have been collected). Therefore, we tend to free multiple nodes at the same time. Out of these, it is usually the case that several of them map to the same memory area. Thus, there is a certain degree of locality in deallocation as well. Hence, if instead of logging every node we unlink from the data structure, we only keep track of the memory areas from which the unlinked nodes come from, we can expect significant savings in term of write-backs to NVRAM: most of the time, the memory area will already be marked as active. We illustrate the potential benefits of the approach in Figure 4.

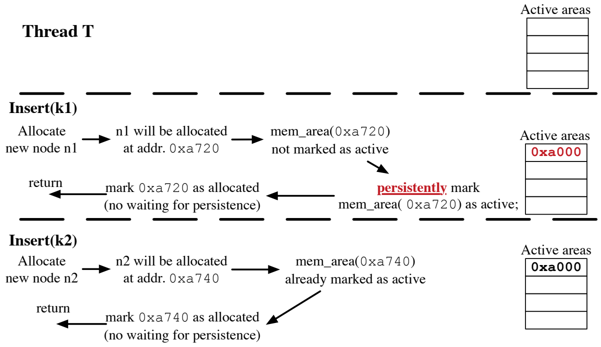

Thread T
Active areas
Insert(k1)
Allocate new node n1
n1 will be allocated at addr. 0xa720
mem_area(0xa720) not marked as active
persistently mark mem_area(0xa720) as active;
mark 0xa720 as allocated (no waiting for persistence)
return
Active areas
0xa000
Insert(k2)
Allocate new node n2
n2 will be allocated at addr. 0xa740
mem_area(0xa740) already marked as active
mark 0xa740 as allocated (no waiting for persistence)
return
Active areas
0xa000

Figure 4: *Illustration: thread T performs two inserts. While both allocate memory, with our approach, only the first allocation performs a durable write.*

While allowing us important time savings at run time, this method does defer some work for when we need to recover. In particular, we need to go over the allocated memory addresses in the active areas at the time of shutdown, and check if they indeed represent nodes that are linked into the data structure. To be able to do this, we also make the assumption data structure nodes belong to memory areas which store no other type of data. To achieve this, we use an allocator specifically for such nodes.

We first briefly describe the principles of the memory reclamation scheme that we employ, after which we go into more details into how we keep track of the active memory page set, and how we recover after a failure.

5.2 Epoch-Based Memory Reclamation

Epoch-based memory reclamation [11] is based on the following principle: if an object is unlinked, then no references to the object are held after the operations concurrent with the unlink have finished. One method of using this principle in practice (and which we use in our reclamation scheme) is to provide each thread with a local counter, keeping track of the *epoch* the current thread finds itself in. The epoch of a thread is incremented when the thread starts an operation, and when it completes it. Thus, if the current epoch number of a thread is odd, the thread is currently active and in the middle of an operation. We collect multiple objects, and free them when the vector formed by the current epochs of the threads is larger than the one when any of the objects were unlinked (only the epochs of threads that were active at the moment of unlinking need to be larger). We refer to the set of unlinked nodes which we attempt to free together as a *generation*.

5.3 Interface with NVRAM Allocators

Memory allocators usually reserve a large contiguous address space, which is then recursively split into smaller chunks. The chunk from which an object is allocated depends on the object's size. These chunks of contiguous memory are generally referred to as allocator *pages*. Since smaller pages from which data structure nodes are directly allocated are part of larger pages, we can configure the granularity of the pages which we keep track of. High-performance concurrent allocators usually partition the memory space for allocations among threads, such that there is minimal communication necessary between them: pages are assigned to individual threads.

Existing persistent allocators provide the capability of atomically allocating and linking (or unlinking and deallocating) objects, which, as discussed, is generally achieved through some form of logging. We do not require this capability: we only require that the persistent allocator is able to correctly maintain its durable metadata when allocating or deallocating. Moreover, in our case, the last write-back (which marks the memory as allocated or free in a thread's local allocator metadata, and is usually the only write-back the allocator issues) does not have to be completed before proceeding: in the case of an allocation, the data structure algorithm will have to wait for the write-backs to complete after the memory is initialized, while in the case of deallocations the memory reclamation scheme waits for all the deallocations it issues at once to be completed. Thus, in most cases, when the allocator only does one store to thread-local data, we do not have to issue a sync operation for the allocator metadata. Based on its metadata and our structures maintaining active pages, the allocator can recover its state in case of a restart. An existing persistent concurrent allocator can be used with our system, with the small changes we mentioned. We also require the allocator to provide a method that returns the next node address to be allocated. As allocators generally assign larger chunks of memory to individual threads, and threads do not "steal" memory from one another, adding this method is trivial. We use a modified version of jemalloc [27], with write-backs inserted when updates to allocator metadata occur to model the run-time performance of persistent allocators.

5.4 Tracking Active Memory Areas

In our approach, each thread keeps a set of active memory pages in a structure called the *active page table (APT)*. For each memory page, we also store some metadata determining when the page can be considered as no longer active and can thus be removed from the set. This metadata consists of (i) the largest epoch at which this thread has unlinked memory belonging to the page from the data structure, and (ii) the largest epoch at which this thread has allocated memory belonging to this page. The addresses of the memory pages need to be stored durably (meaning that when we insert a new page, we have to wait for the write-back of the address of the page to complete before continuing), while the metadata is only needed for removing table entries, and is not needed in case of a restart.

We attempt to trim a thread's active page table when it exceeds a certain size. For this purpose, the metadata associated with each page is used as follows. A page from which unlinks have happened is active until the epoch-based memory reclamation scheme is guaranteed to have freed all unlinked nodes. This can be verified by having the reclamation scheme keep track of the epoch vector of the most recent generation of objects that were collected. A page from which allocations have happened is active until the insert operation has finished, i.e., while the current epoch of the thread is equal to the last epoch at which a node allocation from this page took place. When using a link cache, we also have to ensure that it contains no entries pertaining to the page under consideration. For this reason, the operation that attempts to trim the active page table issues a link cache flush as well. If all the unlinked nodes have been freed, and all the allocated nodes have been linked, the page can be removed from the table.

We use a separate persistent allocator for the active memory page table. Allocations for the table happen very infrequently (we preallocate a number of entries for each thread at start-up, and allocate multiple entries at once when more space is needed; in addition, tables usually do not grow beyond a certain size, and thus no allocations are needed from a certain point). We require that this second allocator provide the interface previous work on NVRAM memory allocators does [6, 24, 42, 51, 56]. In this instance, we used the allocator provided with `nvml` [24].

5.5 Recovery after Transient Failures

On recovery, we must make sure that there are no nodes that are not linked in the data structure but are allocated.

There are two approaches to verifying this, both of which can be parallelized in order to decrease recovery time. The efficiency of each method depends on the size of the data structure, the complexity of the search method, and the size of the memory space that needs to be verified. In both cases, we assume a well-formed data structure. That is, the recovery procedure should first ensure that the data structure is brought to a consistent state before attempting to remove memory leaks. This step is not necessary for any of the data structures we developed.

The first approach is to go over all the node addresses in the active memory pages at the moment of the crash, and, if an address *n* is allocated, search the data structure for *n*'s key. If (i) the search returns a result and (ii) the

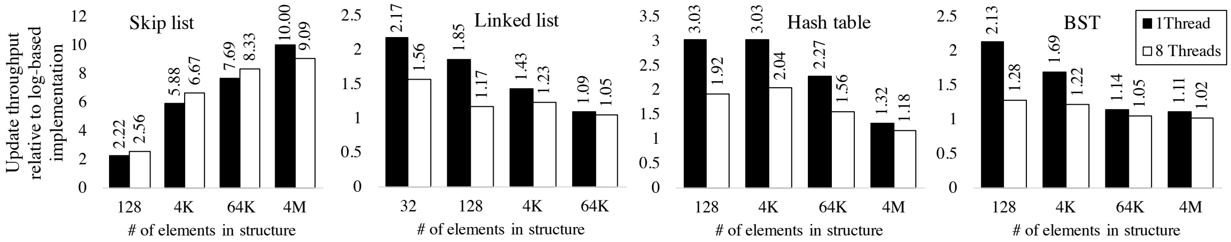


Figure 5: *Update throughput of data structures implemented using our techniques (1 and 8 threads). Values are normalized to throughput of redo log based implementations (1 and 8 threads, respectively).*

address of the returned node is the same as *n*, we leave the node as allocated. Otherwise, we free the node. Condition (ii) is necessary because we might have an allocated but uninitialized node. Therefore, a node with the key that we retrieve from that uninitialized memory might indeed exist in the data structure, but it might not be pointing to this uninitialized memory. The second approach is to traverse the structure only once, and for each node check if its address belongs to the set of active pages. If this is the case, store the address of the node in a volatile memory buffer. Next, go over all the allocated node addresses in the active memory pages, and check if they are in the volatile memory buffer. If they are not, it means they are not linked in the data structure and can be deallocated. Both of these approaches can be parallelized.

We note that in our implementation, it cannot be the case that a node is linked into the data structure, but not marked as allocated. This is because before linking a node in the data structure, we issue a store fence that ensures that the contents of the node, as well as the allocator metadata (for which we issue write-backs, but do not wait for last one to complete when calling the allocation method) are durably written.

6 Evaluation

We now study the impact our proposed techniques have in practice. We look at the overall performance improvements, as well as at the benefits of individual techniques.

6.1 Experimental Setup

We run experiments on an Intel Xeon machine having four E7-4830 v3 12-core processors operating at 2.1–2.7 GHz, with cache sizes of 32KB (L1), 256KB (L2), and 30MB (LLC, per die). We work with key-value pairs, both of which are 8B in size. Nodes are cache-aligned to 64B. Larger values can be accommodated by using indirection instead of directly storing values inside nodes. Shown values are the median of 5 repetitions.

As neither NVRAM with latencies comparable to DRAM, nor processors providing the `clwb` instruction are available yet, we simulate `clwb` by writing data normally, and then pausing for an appropriate number of cycles, similar to previous work [3, 6, 33, 55, 56]. The number of cycles to pause is chosen so as to account for the increased latency of NVRAM (we assume an NVRAM write latency of 125ns, which is an average of the projected values). Intel reports issuing several flushes with `clflushopt` can be up to an order of magnitude faster than flushing them one at a time using `clflush` [22]. We assume similar performance characteristics for the `clwb` instruction. In order to account for the benefit of flushing several cache lines at a time over flushing them one by one, we inject the artifical pause described above only once per batch of cache lines being written to NVRAM (e.g., only once per flush of the link cache).

6.2 Data Structure Performance

We look at the run time behavior of our data structures. We focus on updates, as it is these operations that must be durably recorded in NVRAM. We compare our implementations with alternatives that use lock-based critical sections (and thus use logging). We find that for such data structures, an approach that uses redo logging provides good performance in addition to ensuring durable linearizability. We use the algorithms that we find perform best for each data structure: the lazy linked list [16], a lock-based skip list by Herlihy et al. [18], bst-tk [8], and a hash table with a lazy linked list per bucket. We manually apply logging to each data structure, taking advantage of knowledge of the algorithms so as to minimize the number of *syncs* while maintaining correctness. We do this for fairness of comparison, as the alternative of using a generic transactional/logging framework would have likely resulted in more *syncs* and thus worse performance.

In Figure 5, we show the increase in the number of updates per second obtained by using our structures relative to log-based implementations. We use a workload where 50% of the operations are inserts of random keys, while 50% are removes of random keys, and show results for 1 and 8 concurrent updating threads. We show relative improvements, as the precise latencies are dependent on the assumptions made about `clwb` instruction performance, as well as NVRAM store latencies.

Our method yields important benefits regardless of the data structure type. In particular, for the skip list, where in a log-based implementation a logarithmic number of locks are held while a logarithmic number of updates must be logged, our approach results in an order of magnitude increase in performance. We note that for small

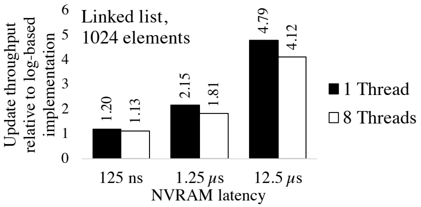


Figure 6: *Update throughput compared to redo log based implementation for various NVRAM write latencies.*

and medium sized data structures, we obtain significant improvements by applying our techniques. For large structures however, our improvements become less impressive. There are two main reasons for this. The first is that as the structure size increases, the latency of an update becomes dominated by the time needed to reach the point in the data structure where the modification needs to be made, both due to the need to traverse more pointers, and because when the structure does not fit in the caches anymore, reads become more expensive. In the case of the linked list in this experiment, it is in fact the only factor that is responsible for the decrease in latency improvement. The second reason has to do with a decrease in the efficiency of our active page tables for deallocations as the structures become large. We discuss why this is, as well as ways of alleviating the issue in § 6.3. In addition, as the number of concurrent updating threads increases, the link cache becomes somewhat less efficient (also discussed in § 6.3). Thus, for high degrees of concurrency, we can turn the link cache off.

In our experiments, we use NVRAM latencies comparable to those of DRAM. Nevertheless, current technologies still have significantly larger latencies. We therefore also perform a simulation where we increase write latency (Figure 6). These measurements are representative of structures which are small enough for reads to be served from cache. As NVRAM write latency increases, our approach becomes much more effective: the ratio between our throughput and that of a log-based implementation becomes inversely proportional to the ratio between the number of *sync* operations in the two approaches.

To summarize, it is important to note that while the precise magnitude of the improvements of our approach may depend on the characteristics of the NVRAM technology being used, these experiments show that our approach is beneficial for all the situations we have considered.

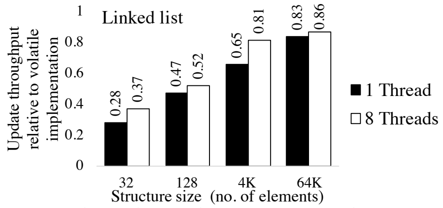


Figure 7: *Update throughput compared to an implementation oblivious of NVRAM.*

We also compare our approach to algorithms aimed at volatile memory, which do not concern themselves with data durability. Briefly, the per-operation overheads related to durability introduced by our approach are constant. While these might account for a non-negligible fraction of the operation latency for small structures, for larger structures, as total operation latency increases, these become less apparent. This is illustrated on a linked list in Figure 7. Thus, in terms of performance, our approach represents a middle ground between volatile data structures and log-based durable approaches.

6.3 Performance: a Closer Look

We now explore the impact each of our techniques has on overall performance.

Link-and-Persist and Link Cache. We evaluate the individual impact on performance of the link-and-persist technique and of the link cache. We measure the throughput of each data structure with the log-based implementation, with a log-free implementation that uses link-and-persist and with a log-free implementation that uses the link cache (all using identical memory management schemes). We then normalize the throughput of the log-free implementations with respect to the log-based implementation to determine the change in performance. We use an update-only workload, with 1024-element data structures. The link cache occupies 32 cache lines.

Figure 8 shows the results. As a result of removing logging, algorithms using link-and-persist outperform log-based alternatives for all structures, both in single-threaded and in concurrent scenarios. Moreover, in most cases, the link cache brings an additional increase in performance with respect to the link-and-persist implementation, due to its batching of write-backs.

NV-Epochs. We evaluate the efficiency of our active page table. The active page table is only efficient if it saves *sync* operations: i.e., if an important fraction of updates do not have to write active page table entries.

We consider 4KB memory pages, and we try to trim an active page table when it exceeds 16 elements. We measure the fraction of allocations and deallocations that

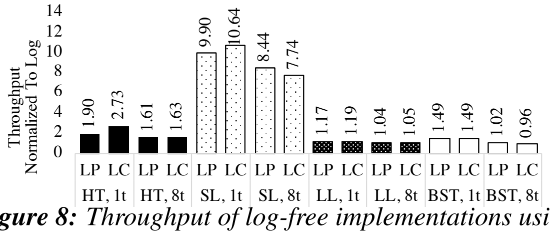


Figure 8: *Throughput of log-free implementations using link-and-persist (LP) and the link cache (LC), normalized to throughput of log-based implementations. Data structures are 1024-element hash table (HT), skip-list (SL), linked list (LL) and binary search tree (BST). Workloads are 100% updates, single-threaded (1t) or with 8 concurrent threads (8t).*

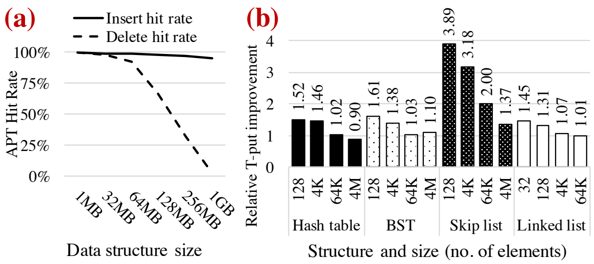


Figure 9: *Active page table hit rates and throughput improvements due to NV-epochs.*

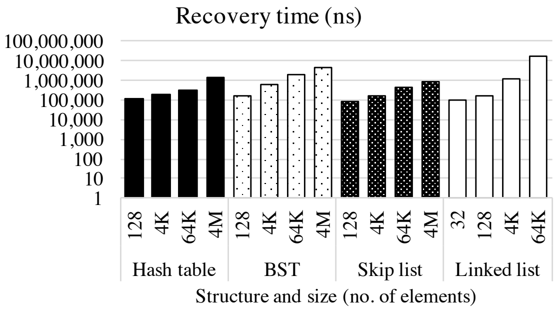


Figure 10: *Data structure recovery times.*

do not need to add an entry to the active page tables (that is, the fraction of hits in the active page table). Results are shown in Figure 9.a. In this experiment, we have used a skip list. Results are similar for other data structures, as the important factor is the data structure size.

We note that the hit rate is close to 100% for allocations, regardless of structure size. In case of deallocations, the hit rate starts decreasing after the structure exceeds 64 MB (more than 1M nodes). This is because as the amount of used memory increases, there is less locality in memory reclamation steps. However, fast memory allocation and deallocation is particularly important for small data structures that fit in the write-back caches, which have small access latencies. In such situations, our approach is effective for both types of operations.

This conclusion is reflected in the throughput observed when using NV-epochs (Figure 9.b): for small and medium-sized structures, NV-epochs can increase throughput several-fold. For large structures, when keeping track of memory at 4KB granularity, NV-epoch's effectiveness decreases. However, the granularity at which we keep track of active memory areas is adjustable. Larger memory areas result in higher hit rates and throughput improvements, at the expense of increased recovery time.

6.4 Recovery

We now measure the time it takes to recover a data structure. We simulate a crash by first stopping execution of the algorithm at an arbitrary point. Then, we ensure the structure's data is not in the write-back caches (by purging the caches). Next, we run the recovery process which first brings the data structure to a consistent state and then traverses its active pages to free allocated-but-not-reachable nodes[3]. We show recovery times for the various structures as a function of their size in Figure 10.

For hash tables, BSTs, and skip-lists, which have fast search methods, recovery is extremely efficient: even in structures with 4M elements, we can recover in less than 5ms. Recovery time for such structures is two orders of magnitude lower than doing a full mark-and-sweep pass in this environment [1]. In the case of the linked list, which has a linear search method, in order to avoid repeated passes over the entire structure, we employ a strategy similar to mark-and-sweep. Recovery in this case is somewhat slower: a linked list with 64K elements can be recovered in 16ms. For all structures, recovery time increases with data structure size. Small structures tend to have a smaller number of active pages at any point in time. In addition, search operations must traverse more pointers for larger structures, and data is less frequently present in the higher-level caches. We believe the observed recovery times are acceptable in case of a reboot.

6.5 NV-Memcached

We now show how our techniques can be applied in a larger context by developing an object caching system for durable memory: *NV-Memcached*. The main idea behind *NV-Memcached* is to make Memcached durable by replacing its core data structures—the hash table and the slab allocator—with durable versions. This transformation entails interesting technical challenges.

First, Memcached uses a lock-protected sequential hash table; thus replacing it with our durable non-blocking hash table would negate the latter's lock-freedom. We solve this challenge by basing *NV-Memcached* on *memcached-clht* [34], a version of Memcached that avoids protecting the hash table with locks by employing a concurrent hash table—CLHT [8], and replacing CLHT with our log-free durable hash table.

The second challenge is related to the recovery of items. With a naive implementation of a durable slab allocator, it is possible for memory leaks to occur after a restart. An item can be allocated, but not yet linked in the hash table, or an item can be unlinked from the hash table but not yet marked as free in the allocator. We address this issue with a similar approach to our active page technique (§ 5): we keep track of active slabs. During recovery, we iterate over each thread's active slab table and free any memory which is marked as allocated but not yet or no longer reachable from the hash table.

We compare the performance of *NV-Memcached* to that of Memcached and memcached-clht using

[3] After recovery, new threads can spawn and resume execution at a "safe" point (a point in the instruction stream from which execution can continue regardless of when the crash occurred). Determining such safe points in general is outside the scope of this paper, but for our specific case, any point in-between two data structure operations is a safe point.

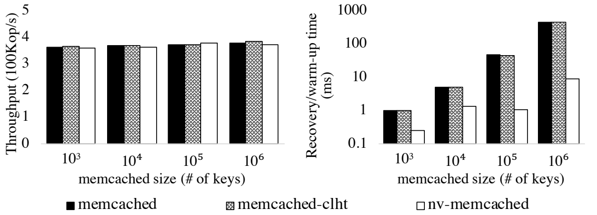


Figure 11: Performance and warm-up time comparison of Memcached and NV-Memcached.

memtier-benchmark [50]. The benchmark runs for a predetermined amount of time, issuing a mix of `get` and `set` operations using keys drawn uniformly at random from a given key range. The key range and the ratio of `get` to `set` operations are configurable. Before each experiment, we warm up the cache by inserting items covering half of the key range. Both the server and the client are run with the default number of threads (4). The results are averaged over 5 runs for each configuration.

The first experiment compares the throughput of the three systems for different key ranges, under a 1:4 `set` to `get` ratio. Figure 11 shows that there is no notable performance drop between Memcached, memcached-clht and *NV-Memcached.* Thus, our techniques remain practical when applied to real-world applications.

The second experiment compares, for three different key range sizes, the warm-up time of Memcached and memcached-clht (the time it takes to populate the cache with half of the key range) to the recovery time of *NV-Memcached* (the time it takes to recover after a restart). Figure 11 shows that populating a (volatile) Memcached or memcached-clht instance with items can take up to three orders of magnitude more time than recovering a *NV-Memcached* instance of the same size. This justifies the practicality of a non-volatile memory caching service—recovering such a service after a machine restart takes just a fraction of the time necessary for its volatile counterpart to get re-populated (and thus be useful again).

7 Related Work

Several approaches have used transactions as a means of interaction with NVRAM [2, 6, 9, 14, 25, 28, 30, 33, 37, 56]. The benefits of transaction-based approaches are generality and ease of use. Yet, their inherent and significant overhead has been recently highlighted (e.g., [52]), and several attempts to alleviate the problem have been proposed. Izraelevitz et al. [25] introduce an approach in which by reliably keeping track of the last executed store instruction at each thread, one is able to simply complete the execution of critical sections after a restart. While efficient if write-back caches are persistent, the approach otherwise requires a write to the log for each store in critical sections. Kolli et al. [30] focus on static transactions in lock-based applications, and attempt to minimize persist dependencies in order to limit waiting time. The authors also show how the commit stage of transactions can be performed while not holding any locks. Similarly, Kamino-Tx [37] uses a copy-on-write technique, and avoids logging in critical sections. DudeTM [33] optimizes redo logging by first executing the transaction and obtaining a redo log in volatile memory, then atomically flushing the redo log to persistent memory, and only then modifying the original data.

In this paper, we go beyond optimizations to logging: we provide a method that in the common case when locality is preserved, allows us to circumvent such logging altogether in the context of concurrent data structures.

A number of efforts [2, 4, 20] have been dedicated to the generation of correct durable applications for NVRAM from existing code. These approaches generally assume lock-based code. Due to their general-purpose nature, they incur additional overheads when compared to our method, in particular due to logging.

Several proposals for indexing trees for NVRAM have been made [5, 32, 48, 54, 58]. However, they either require logging in some form, or do not address potential memory leaks during new node creation. In addition, the techniques cannot be easily generalized to other data structures. Friedman et al. [12] introduce lock-free algorithms for durable queues, but do not go beyond this data structure, or consider memory management. Other work advocating lock-free algorithms either assumes durable caches [46, 47], or automatically generates durable algorithms that issue syncs after each update [26].

The problem of general memory allocation and reclamation for NVRAM has also received attention. Generic persistent memory frameworks [2, 6, 35, 56] handle allocation and reclamation as part of the transaction mechanisms they provide, and thus rely on logging. nvm_malloc [51] provides an interface to allocate and free persistent memory, but because of fine-grained accounting, incurs significant overheads for each allocation and deallocation. Makalu [1] and NVthreads [20] also keep track of allocator metadata at a coarser-grain level. However, they incur higher costs at recovery time, as they require a garbage collection pass over the entire memory. Unlike all these approaches, we propose a method that is highly tuned to concurrent data structures. Thus, we are able to minimize overheads at both run time and recovery time. Our approach in fact builds upon basic memory allocators, and handles concurrent memory reclamation as well. Thus, our scheme can take advantage of an efficient durable memory allocator at its core.

Other Memcached adaptations for NVRAM have been proposed, but they use transactions extensively [6, 36, 44] or they do not guarantee all completed requests are durable [57], whereas *NV-Memcached* ensures all completed requests are durable and limits transactions to the slab allocator code by using our log-free hash table.

References

[1] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Makalu: Fast Recoverable Allocation of Non-Volatile Memory. OOPSLA 2016.

[2] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. OOPSLA 2014.

[3] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. Rewind: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. VLDB 2015.

[4] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. NVMove: Helping Programmers Move to Byte-Based Persistence. INFLOW 2016.

[5] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. VLDB 2015.

[6] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. ASPLOS 2011.

[7] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The Inherent Cost of Remembering Consistently. SPAA 2018.

[8] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. ASPLOS 2015.

[9] Joel Edward Denny, Seyong Lee, and Jeffrey S. Vetter. Language-Based Optimizations for Persistence on Nonvolatile Main Memory Systems. IPDPS 2017.

[10] Facebook. RocksDB. `http://rocksdb.org`.

[11] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004.

[12] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A Persistent Lock-Free Queue for Non-Volatile Memory. PPoPP 2018.

[13] Google. LevelDB. `http://leveldb.org`.

[14] Yonatan Gottesman, Joel Nider, Ronen Kat, Yaron Weinsberg, and Michael Factor. Using Storage Class Memory Efficiently for an In-Memory Database. SYSTOR 2016.

[15] Timothy L Harris. A Pragmatic Implementation of Non-Blocking Linked Lists. DISC 2001.

[16] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. OPODIS 2005.

[17] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[18] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A Simple Optimistic Skiplist Algorithm. SIROCCO 2007.

[19] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[20] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-Threaded Applications. EuroSys 2017.

[21] Intel. Intel Architecture Instruction Set Extensions Programming Reference. `https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf`.

[22] Intel. Intel64 and IA-32 Architectures Optimization Reference Manual. `https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`.

[23] Intel. Intel64 and IA-32 Architectures Software Developers Manuals Combined. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[24] Intel. NVM Library. `http://pmem.io`.

[25] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. ASPLOS 2016.

[26] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. DISC 2016.

[27] jemalloc. `http://jemalloc.net/`.

[28] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. ASPLOS 2016.

[29] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-Level Persistency. ISCA 2017.

[30] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-Performance Transactions for Persistent Memories. ASPLOS 2016.

[31] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. ISCA 2009.

[32] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. FAST 2017.

[33] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DUDETM: Building Durable Transactions with Decoupling for Persistent Memory. ASPLOS 2017.

[34] LPD-EPFL. memcached-clht. `https://github.com/LPD-EPFL/memcached-clht`.

[35] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. Persistent Memory Transactions. *arXiv preprint arXiv:1804.00701*, 2018.

[36] Virendra Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. HotStorage 2017.

[37] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx. EuroSys 2017.

[38] Memcached. `http://www.memcached.org`.

[39] Micron. 3D XPoint Technology. https://www.micron.com/about/our-innovation/3d-xpoint-technology.

[40] MonetDB. http://www.monetdb.org.

[41] MongoDB. http://www.mongodb.org.

[42] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, Durable, and Safe Memory Management for Byte-Addressable Non-Volatile Main Memory. TRIOS 2013.

[43] MySQL. http://www.mysql.com.

[44] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. ASPLOS 2017.

[45] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-Free Binary Search Trees. PPoPP 2014.

[46] Faisal Nawab, Dhruva Chakrabarti, Terrence Kelly, and Charles B. Morrey. Zero-Overhead NVM Crash Resilience. NVMW 2015.

[47] Faisal Nawab, Dhruva R Chakrabarti, Terence Kelly, and Charles B Morrey III. Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience. Technical Report HPL-2014-70. Hewlett-Packard, 2014.

[48] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. SIGMOD 2016.

[49] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology.

[50] Redis Labs. NoSQL Redis and Memcache Traffic Generation and Benchmarking Tool. https://github.com/RedisLabs/memtier_benchmark.

[51] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory Allocation for NVRAM. ADMS 2015.

[52] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the Long Latency of Persist Barriers Using Speculative Execution. ISCA 2017.

[53] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The Missing Memristor Found. *Nature*, 453(7191):80, 2008.

[54] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. FAST 2011.

[55] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. Aerie: Flexible File-System Interfaces to Storage-Class Memory. EuroSys 2014.

[56] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight Persistent Memory. ASPLOS 2011.

[57] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. NVMcached: An NVM-based Key-Value Cache. APSys 2016.

[58] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. FAST 2015.

[59] Yiying Zhang and Steven Swanson. A Study of Application Performance with Non-Volatile Main Memory. MSSR 2015.

Stable and Consistent Membership at Scale with Rapid

Lalith Suresh[†], Dahlia Malkhi[†], Parikshit Gopalan[†], Ivan Porto Carreiro[◇], Zeeshan Lokhandwala[‡]

[†]VMware Research, [‡]VMware, [◇]One Concern

Abstract

We present the design and evaluation of Rapid, a distributed membership service. At Rapid's core is a scheme for *multi-process cut detection* (CD) that revolves around two key insights: (i) it suspects a failure of a process only after alerts arrive from multiple sources, and (ii) when a group of processes experience problems, it detects failures of the entire group, rather than conclude about each process individually. Implementing these insights translates into a simple membership algorithm with low communication overhead.

We present evidence that our strategy suffices to drive unanimous detection almost-everywhere, even when complex network conditions arise, such as one-way reachability problems, firewall misconfigurations, and high packet loss. Furthermore, we present both empirical evidence and analyses that proves that the almost-everywhere detection happens with high probability. To complete the design, Rapid contains a leaderless consensus protocol that converts multi-process cut detections into a view-change decision. The resulting membership service works both in fully decentralized as well as logically centralized modes.

We present an evaluation of Rapid in moderately scalable cloud settings. Rapid bootstraps 2000 node clusters 2-5.8x faster than prevailing tools such as Memberlist and ZooKeeper, remains stable in face of complex failure scenarios, and provides strong consistency guarantees. It is easy to integrate Rapid into existing distributed applications, of which we demonstrate two.

1 Introduction

Large-scale distributed systems today need to be provisioned and resized quickly according to changing demand. Furthermore, at scale, failures are not the exception but the norm [21, 30]. This makes membership management and failure detection a critical component of any distributed system.

Our organization ships standalone products that we do not operate ourselves. These products run in a wide range of enterprise data center environments. In our experience, many failure scenarios are not always crash failures, but commonly involve misconfigured firewalls, one-way connectivity loss, flip-flops in reachability, and some-but-not-all packets being dropped (in line with observations by [49, 19, 37, 67, 41]). We find that existing membership solutions struggle with these common failure scenarios, despite being able to cleanly detect crash faults. In particular, existing tools take long to, or never converge to, a stable state where the faulty processes are removed (§2.1).

We posit that despite several decades of research and production systems, stability and consistency of existing membership maintenance technologies remains a challenge. In this paper, we present the design and implementation of *Rapid*, a scalable, distributed membership system that provides both these properties. We discuss the need for these properties below, and present a formal treatment of the service guarantees we require in §3.

Need for stability. Membership changes in distributed systems trigger expensive recovery operations such as failovers and data migrations. Unstable and flapping membership views therefore cause applications to repeatedly trigger these recovery workflows, thereby severely degrading performance and affecting service availability. This was the case in several production incidents reported in the Cassandra [10, 9] and Consul [26, 25, 27] projects. In an end-to-end experiment, we also observed a 32% increase in throughput when replacing a native system's failure detector with our solution that improved stability (see §7 for details).

Furthermore, failure recovery mechanisms may be faulty *themselves* and can cause catastrophic failures when they run amok [42, 40]. Failure recovery workflows being triggered ad infinitum have led to Amazon EC2 outages [5, 6, 4], Microsoft Azure outages [7, 48], and "killer bugs" in Cassandra and HBase [39].

Given these reasons, we seek to avoid frequent oscillations of the membership view, which we achieve through stable failure detection.

Need for consistent membership views. Many systems require coordinated failure recovery, for example, to correctly handle data re-balancing in storage systems [3, 28]. Consistent changes to the membership view simplify reasoning about system behavior and the development of dynamic reconfiguration mechanisms [65].

Conversely, it is challenging to build reliable clustered services on top of a weakly consistent membership service [11]. Inconsistent view-changes may have detrimental effects. For example, in sharded systems that rely on consistent hashing, an inconsistent view of the cluster leads to clients directing requests to servers that do not host the relevant keys [12, 3]. In Cassandra, the lack of consistent membership causes nodes to duplicate data re-balancing efforts when concurrently adding nodes to a cluster [11] and also affects correctness [12]. To work around the lack of consistent membership, Cassandra ensures that only a single node is joining the cluster at any given point in time, and operators are advised to wait *at least two minutes* between adding each new node to a cluster [11]. As a consequence, bootstrapping a 100 node Cassandra cluster takes three hours and twenty minutes, thereby significantly slowing down provisioning [11].

For these reasons, we seek to provide strict consistency, where membership changes are driven by agreement among processes. Consistency adds a layer of safety above the failure detection layer and guarantees the same membership view to all non-faulty processes.

Our approach

Rapid is based on the following fundamental insights that bring stability and consistency to *both* decentralized and logically centralized membership services:

Expander-based monitoring edge overlay. To scale monitoring load, Rapid organizes a set of processes (a *configuration*) into a stable failure detection topology comprising *observers* that monitor and disseminate reports about their communication *edges* to their *subjects*. The monitoring relationships between processes forms a directed expander graph with strong connectivity properties, which ensures with a high probability that healthy processes detect failures. We interpret multiple reports about a subject's edges as a high-fidelity signal that the subject is faulty.

Multi-process cut detection. For stability, processes in Rapid (i) suspect a faulty process p only upon receiving alerts from multiple observers of p, and (ii) delay acting on alerts about different processes until the churn stabilizes, thereby converging to detect a global, possibly multi-node *cut* of processes to add or remove from the membership. This filter is remarkably simple to implement, yet it suffices by itself to achieve *almost-everywhere agreement* – unanimity among a large fraction of processes about the detected cut.

Practical consensus. For consistency, we show that converting almost-everywhere agreement into full agreement is practical even in large-scale settings. Rapid's consensus protocol drives configuration changes by a low-overhead, leaderless protocol in the common case: every process simply validates consensus by counting the number of identical cut detections. If there is a quorum containing three-quarters of the membership set with the same cut, then without a leader or further communication, this is a safe consensus decision.

Rapid thereby ensures all participating processes see a strongly consistent sequence of membership changes to the cluster, while ensuring that the system is stable in the face of a diverse range of failure scenarios.

In summary, we make the following key contributions:

- Through measurements, we demonstrate that prevailing membership solutions guarantee neither stability nor consistency in the face of complex failure scenarios.
- We present the design of Rapid, a scalable membership service that is robust in the presence of diverse failure scenarios while providing strong consistency. Rapid runs both as a decentralized as well as a logically centralized membership service.
- In system evaluations, we demonstrate how Rapid, despite offering much stronger guarantees, brings up 2000 node clusters 2-5.8x faster than mature alternatives such as Memberlist and ZooKeeper. We demonstrate Rapid's robustness in the face of different failure scenarios such as simultaneous node crashes, asymmetric network partitions and heavy packet loss. Rapid achieves these goals at a similar cost to existing solutions.
- Lastly, we report on our experience running Rapid to power two applications; a distributed transactional data platform and a service discovery use case.

2 Motivation and Related work

Membership solutions today fall into two categories. They are either managed for a cluster through an auxiliary service [15, 43], or they are gossip-based and fully decentralized [45, 44, 8, 69, 62, 59, 70, 64].

We studied how three widely adopted systems behave in the presence of network failure scenarios: *(i)* of the first category, ZooKeeper [15], and of the second, *(ii)* Memberlist [47], the membership library used by Consul [45] and Serf [44] and *(iii)* Akka Cluster [69] (see §7 for the detailed setup). For ZooKeeper and Memberlist, we bootstrap a 1000 process cluster with standalone agents that join and maintain membership using these solutions (for Akka Cluster, we use 400 processes because it began failing for cluster sizes beyond 500). We then drop 80% of packets for 1% of processes, simulating high packet loss scenarios described in the literature [19, 49] that we have also observed in practice.

Figure 1 shows a timeseries of membership sizes, as viewed by each non-faulty process in the cluster (every dot indicates a single measurement by a process). Akka Cluster is unstable as conflicting rumors about processes propagate in the cluster concurrently, even resulting in benign processes being removed from the membership. Memberlist and ZooKeeper resist removal of the faulty processes from the membership set but are unstable over

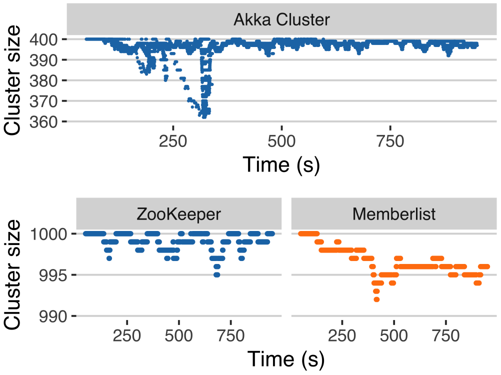


Figure 1: Akka Cluster, ZooKeeper and Memberlist exhibit instabilities and inconsistencies when 1% of processes experience 80% packet loss (similar to scenarios described in [19, 49]). Every process logs its own view of the cluster size every second, shown as one dot along the time (X) axis. Note, the y-axis range does not start at 0. X-axis points (or intervals) with different cluster size values represent inconsistent views among processes at that point (or during the interval).

a longer period of time. We also note extended periods of inconsistencies in the membership view.

Having found existing membership solutions to be unstable in the presence of typical network faults, we now proceed to discuss the broader design space.

2.1 Comparison of existing solutions

There are three membership service designs in use today, each of which provides different degrees of resiliency and consistency.

Logically centralized configuration service. A common approach to membership management in the industry is to store the membership list in an auxiliary service such as ZooKeeper [15], etcd [33], or Chubby [23].

The main advantage of this approach is simplicity: a few processes maintain the ground truth of the membership list with strong consistency semantics, and the remaining processes query this list periodically.

The key shortcoming here is that relying on a small cluster reduces the overall resiliency of the system: connectivity issues to the cluster, or failures among the small set of cluster members themselves, may render the service unavailable (this led Netflix to build solutions like Eureka [58, 61]). As the ZooKeeper developers warn, this also opens up new failure modes for applications that depend on an auxiliary service for membership [17].

Gossip-based membership. van Renesse et al. [72, 71] proposed managing membership by using gossip to spread positive notifications (keepalives) between all processes. If a process p fails, other processes eventually remove p after a timeout. SWIM [29] was proposed as a variant of that approach that reduces the communication overhead; it uses gossip to spread "negative" alerts, rather than regular positive notifications.

Gossip-based membership schemes are widely adopted in deployed systems today, such as Cassandra [8], Akka [69], ScyllaDB [64], Serf [44], Redis Cluster [62], Orleans [59], Uber's Ringpop [70], Netflix's Dynomite [56], and some systems at Twitter [55].

The main advantage of gossip-based membership is resiliency and graceful degradation (they tolerate $N-1$ failures). The key disadvantages include their weak consistency guarantees and the complex emergent behavior that leads to stability problems.

Stability is a key challenge in gossip-based membership: When communication fails between two processes which are otherwise live and correct, there are repeated accusations and refutations that may cause oscillations in the membership views. As our investigation of leading gossip-based solutions showed (Figure 1), these conflicting alerts lead to complex emergent behavior, making it challenging to build reliable clustered services on top of. Indeed, stability related issues with gossip are also observed in production settings (see, e.g., Consul [26, 25, 27] and Cassandra [11, 12, 10]).

Lastly, FireFlies [50] is a decentralized membership service that tolerates Byzantine members. FireFlies organizes monitoring responsibilities via a randomized k-ring topology to provide a robust overlay against Byzantine processes. While the motivation in FireFlies was different, we believe it offers a solution for stability; accusations about a process by a potentially Byzantine monitor are not acted upon until a conservative, fixed delay elapses. If a process does not refute an accusation about it within this delay, it is removed from the membership. However, the FireFlies scheme is based on a gossip-style protocol involving accusations, refutations, rankings, and disabling (where a process p announces that a monitor should not complain about it). Furthermore, FireFlies' refutations resist process removals as much as possible, which is undesirable in non-Byzantine settings. For example, in the 80% packet loss scenario described in Figure 1, a faulty process p may still succeed in disseminating refutations, thereby resisting removal from the membership. As we show in upcoming sections, our scheme is simple in comparison and requires little bookkeeping per process. Unlike FireFlies, we aggregate reports about a process p from multiple sources to decide whether to remove p, enabling timely and coordinated membership changes with low overhead.

Group membership. By themselves, gossip-based membership schemes do not address consistency, and allow the membership views of processes to diverge. In this sense, they may be considered more of failure detectors, than membership services.

Maintaining membership with strict consistency guarantees has been a focus in the field of fault tolerant state-machine replication (SMR), starting with early foundations of SMR [52, 60, 63], and continuing with a variety of group communication systems (see [24] for a survey of GC works). In SMR systems, membership is typically needed for selecting a unique primary and for enabling dynamic service deployment. Recent work on Census [28] scales dynamic membership maintenance to a locality-aware hierarchy of domains. It provides fault tolerance by running the view-change consensus protocol only among a sampled subset of the membership set.

These methods may be harnessed on top of a stable failure detection facility, stability being orthogonal to the consistency they provide. As we show, our solution uses an SMR technique that benefits from stable failure detection to form fast, leaderless consensus.

3 The Rapid Service

Our goal is to create a membership service based on techniques that apply equally well to both decentralized as well as logically centralized designs. For ease of presentation, we first describe the fully decentralized Rapid service and its properties in this section, followed by its design in §4. We then relax the resiliency properties in §5 for the logically centralized design.

API Processes use the membership service by using the Rapid library and invoking a call JOIN(HOST:PORT, SEEDS, VIEW-CHANGE-CALLBACK). Here, HOST:PORT is the process' TCP/IP listen address. Internally, the join call assigns a unique logical identifier for the process (ID). If a process departs from the cluster either due to a failure or by voluntarily leaving, it rejoins with a new ID. This ID is internal to Rapid and is not an identifier of the application that is using Rapid. SEEDS is an initial set of process addresses known to everyone and used to contact for bootstrapping. VIEW-CHANGE-CALLBACK is used to notify applications about membership change events.

Configurations A configuration in Rapid comprises a configuration identifier and a *membership-set* (a list of processes). Each process has a local *view* of the configuration. All processes use the initial seed-list as a bootstrap configuration C_0. Every configuration change decision triggers an invocation of the VIEW-CHANGE-CALLBACK at all processes, that informs processes about a new configuration and membership set.

At time t, if C is the configuration view of a majority of its members, we say that C is the *current configuration*. Initially, once a majority of C_0 start, it becomes current.

Failure model We assume that every pair of correct processes can communicate with each other within a known transmission delay bound (an assumption required for failure detection). When this assumption is violated for a pair of (otherwise live) processes, there is no obvious definition to determine which one of them is faulty (though at least one is). We resolve this using the parameters L and K as follows. Every process p (a *subject*) is monitored by K *observer* processes. If L-of-K correct observers cannot communicate with a subject, then the subject is considered *observably unresponsive*. We consider a process faulty if it is either crashed or observably unresponsive.

Cut Detection Guarantees Let C be the current configuration at time t. Consider a subset of processes $F \subset C$ where $\frac{|F|}{|C|} < \frac{1}{2}$. If all processes in $C \setminus F$ remain non-faulty, we guarantee that the multi-process cut will eventually be detected and a view-change $C \setminus F$ installed [1]:

- Multi-process cut detection: With high probability, every process in $C \setminus F$ receives a multi-process cut detection notification about F. In Rapid, the probability is taken over all the random choices of the observer/subject overlay topology, discussed in §4. The property we use is that with high probability the underlying topology remains an expander at all times, where the expansion is quantified in terms of its second eigenvalue.

A similar guarantee holds for joins. If at time t a set J of processes join the system and remain non-faulty, then every process in $C \cup J$ is notified of J joining.

Joins and removals can be combined: If a set of processes F as above fails, and a set of processes J joins, then $(C \setminus F) \cup J$ is eventually notified of the changes.

- View-Change: Any view-change notification in C is by consensus, maintaining *Agreement* on the view-change membership among all correct processes in C; and *Liveness*, provided a majority of $C \cup J$ ($J = \emptyset$ if there are no joiners) remain correct until the VC configuration becomes current.

Our requirements concerning configuration changes hold when the system has quiesced. During periods of instability, intermediate detection(s) may succeed, but there is no formal guarantee about them.

Hand-off Once a new configuration C_{j+1} becomes current, we abstractly abandon C_j and start afresh: New failures can happen within C_{j+1} (for up to half of the membership set), and the Cut Detection and View Change guarantees must hold.

We note that liveness of the consensus layer depends on a majority of both C_j and C_{j+1} remaining correct to perform the 'hand-off': Between the time when C_j becomes current and until C_{j+1} does, no more than a minority fail in either configuration. This dynamic model

[1] The size of cuts $|F|$ we can detect is a function of the monitoring topology. The proof is summarized in §8, and a full derivation appears in a tech report [51].

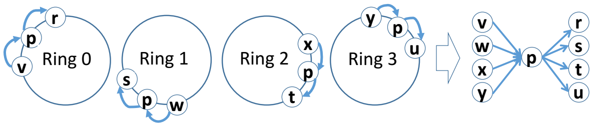


Figure 2: p's neighborhood in a $K = 4$-Ring topology. p's observers are $\{v,w,x,y\}$; p's subjects are $\{r,s,t,u\}$.

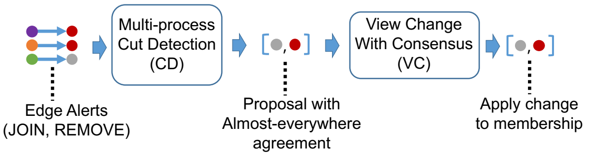

Multi-process Cut Detection (CD)
View Change With Consensus (VC)
Edge Alerts (JOIN, REMOVE)
Proposal with Almost-everywhere agreement
Apply change to membership

Figure 3: Solution overview, showing the sequence of steps at each process for a single configuration change.

borrows the dynamic-interplay framework of [35, 66].

4 Decentralized Design

Rapid forms an immutable sequence of configurations driven through consensus decisions. Each configuration may drive a single configuration-change decision; the next configuration is logically a new system as in the virtual synchrony approach [22]. Here, we describe the algorithm for changing a known current configuration $\mathscr{C}$, consisting of a membership set (a list of process identities). When clear from the context, we omit $\mathscr{C}$ or explicit mentions of the configuration, as they are fixed within one instance of the configuration-change algorithm.

We start with a brief overview of the algorithm, breaking it down to three components: (1) a monitoring overlay; (2) an almost-everywhere multi-process cut detection (CD); and (3) a fast, leaderless view-change (VC) consensus protocol. The response to problems in Rapid evolves through these three components (see Figure 3).

Monitoring We organize processes into a monitoring topology such that every process monitors K peers and is monitored by K peers. A process being monitored is referred to as a *subject* and a process that is monitoring a subject is an *observer* (each process therefore has K subjects and K observers). The particular topology we employ in Rapid is an *expander graph* [38] realized using K pseudo-random rings [34]. Other observer/subject arrangements may be plugged into our framework without changing the rest of the logic.

Importantly, this topology is deterministic over the membership set $\mathscr{C}$; every process that receives a notification about a new configuration locally determines its subjects and creates the required monitoring channels.

There are two types of alerts generated by the monitoring component, REMOVE and JOIN. A REMOVE alert is broadcast by an observer when there are reachability problems to its subject. A JOIN alert is broadcast by an observer when it is informed about a subject joiner request. In this way, both types of alerts are generated by multiple sources about the same subject. Any best-effort broadcast primitive may be used to disseminate alerts (we use gossip-based broadcast).

Multi-process cut detection (CD) REMOVE and JOIN alerts are handled at each process independently by a multi-process cut detection (CD) mechanism. This mechanism collects evidence to support a single, stable multi-process configuration change proposal. It outputs the same cut proposal *almost-everywhere*; i.e., unanimity in the detection among a large fraction of processes.

The CD scheme with a K-degree monitoring topology has a constant per-process per-round communication cost, and provides stable multi-process cut detection with almost-everywhere agreement.

View change (VC) Finally, we use a consensus protocol that has a fast path to agreement on a view-change. If the protocol collects identical CD proposals from a *Fast Paxos quorum* (three quarters) of the membership, then it can decide in one step. Otherwise, it falls back to Paxos to form agreement on some proposal as a view-change.

We note that other consensus solutions could use CD as input and provide view-change consistency. VC has the benefit of a fast path to decision, taking advantage of the identical inputs almost-everywhere.

We now present a detailed description of the system.

4.1 Expander-based Monitoring

Rapid organizes processes into a monitoring topology that is an *expander graph* [38]. Specifically, we use the fact that a random K-regular graph is very likely to be a good expander for $K \geq 3$ [34]. We construct K pseudo-randomly generated rings with each ring containing the full list of members. A pair of processes (o, s) form an observer/subject *edge* if o precedes s in a ring. Duplicate edges are allowed and will have a marginal effect on the behavior. Figure 2 depicts the neighborhood of a single process p in a 4-Ring topology.

Topology properties. Our monitoring topology has three key properties. The first is *expansion*: the number of edges connecting two sets of processes reflects the relative sizes of the set. This means that if a small subset F of processes V are faulty, we should see roughly $\frac{|V|-|F|}{|V|}$ fraction of monitoring edges to F emanating from the set $V \setminus F$ of healthy processes. This ensures with high probability that healthy processes detect failures, as long as the set of failures is not too large. The size of failures we can detect depends on the expansion of the topology as quantified by the value of its second eigenvalue (§ 8). Second, every process monitors K subjects, and is monitored by K observers. Hence, monitoring incurs $O(K)$ overhead

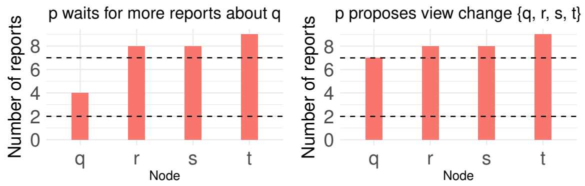


Figure 4: Almost everywhere agreement protocol example at a process p, with tallies about q,r,s,t and $K = 10, H = 7, L = 2$. K is the number of observers per subject. The region between H and L is the unstable region. The region between K and H is the stable region. **Left:** $stable = \{r,s,t\}$; $unstable = \{q\}$. **Right:** q moves from $unstable$ to $stable$; p proposes a view change $\{q,r,s,t\}$.

per process per round, distributing the load across the entire cluster. The fixed observer/subject approach distinguishes Rapid from gossip-based techniques, supporting prolonged monitoring without sacrificing failure detection scalability. At the same time, we compare well with the overhead of gossip-based solutions (§7). Third, every process join or removal results only in $2 \cdot K$ monitoring edges being added or removed.

Joins New processes join by contacting a list of K temporary observers obtained from a seed process (deterministically assigned for each joiner and $\mathscr{C}$ pair, until a configuration change reflects the join). The temporary observers generate independent alerts about joiners. In this way, multiple JOIN alerts are generated from distinct sources, in a similar manner to alerts about failures.

Pluggable edge-monitor. A monitoring edge between an observer and its subject is a pluggable component in Rapid. With this design, Rapid can take advantage of diverse failure detection and monitoring techniques, e.g., history-based adaptive techniques as used by popular frameworks like Hystrix [57] and Finagle [68]; phi-accrual failure detectors [31]; eliciting indirect probes [29]; flooding a suspicion and allowing a timeout period for self-rebuttal [50]; using cross-layer information [54]; application-specific health checks; and others.

Irrevocable Alerts When the edge-monitor of an observer indicates an existing subject is non-responsive, the observer broadcasts a REMOVE alert about the subject. Given the high fidelity made possible with our stable edge monitoring, these alerts are considered *irrevocable*, thus Rapid prevents spreading conflicting reports. When contacted by a subject, a temporary observer broadcasts JOIN alert about the subject.

4.2 Multi-process Cut Detection

Alerts in Rapid may arrive at different orders at each process. Every process independently aggregates these alerts until a stable multi-process cut is detected. Our approach aims to reach agreement *almost everywhere* with regards to this detection. Our mechanism is based on a simple key insight: A process defers a decision on a single process until the alert-count it received on all processes is considered stable. In particular, it waits until there is no process with an alert-count above a low-watermark threshold L and below a high-watermark threshold H.

Our technique is simple to implement; it only requires maintaining integer counters per-process and comparing them against two thresholds. This state is reset after each configuration change.

Processing REMOVE and JOIN alerts Every process ingests broadcast alerts by observers about edges to their subjects. A REMOVE alert reports that an edge to the subject process is faulty; a JOIN alert indicates that an edge to the subject is to be created. By design, a JOIN alert can only be about a process not in the current configuration $\mathscr{C}$, and REMOVE alerts can only be about processes in $\mathscr{C}$. There cannot be JOIN and REMOVE alerts about the same process in $\mathscr{C}$.

Every process p tallies up distinct REMOVE and JOIN alerts in the current configuration view as follows. For each observer/subject pair (o, s), p maintains a value $M(o,\ s)$ which is set to 1 if an alert was received from observer o regarding subject s; and it is set to (default) 0 if no alert was received. A *tally(s)* for a process s is the sum of entries $M(*,\ s)$.

Stable and unstable report modes We use two parameters H and L, $1 \leq L \leq H \leq K$. A process p considers a process s to be in a *stable report mode* if $|\text{tally}(s)| \geq H$ at p. A stable report mode indicates that p has received at least H distinct observer alerts about s, hence we consider it "high fidelity"; A process s is in an *unstable report mode* if $\text{tally}(s)$ is in between L and H. If there are fewer than L distinct observer alerts about s, we consider it noise. Recall that Rapid does not revert alerts; hence, a stable report mode is permanent once it is reached. Note that, the same thresholds are used for REMOVE and JOIN reports; this is not mandatory, and is done for simplicity.

Aggregation Each process follows one simple rule for aggregating tallies towards a proposed configuration change: *delay proposing a configuration change until there is at least one process in stable report mode and there is no process in unstable report mode*. Once this condition holds, the process announces a configuration change proposal consisting of all processes in stable report mode, and the current configuration identifier. The proposed configuration change has the almost-everywhere agreement property, which we analyze in §8 and evaluate in §7. Figure 4 depicts the almost everywhere agreement mechanism at a single process.

Ensuring liveness: implicit detections and reinforcements There are two cases in which a subject process might get stuck in an unstable report mode and not accrue H observer reports. The first is when the observers themselves are faulty. To prevent waiting for stability forever, for each observer o of s, if both o and s are in the unstable report mode, then an *implicit-alert* is applied from o to s (i.e., an implicit REMOVE if s is in $\mathscr{C}$ and a JOIN otherwise; o is by definition always in $\mathscr{C}$).

The second is the case when a subject process has good connections to some observers, and bad connections to others. In this case, after a subject s has been in the unstable reporting mode for a certain timeout period, each observer o of s *reinforces* the detection: if o did not send a REMOVE message about s already, it broadcasts a REMOVE about s to echo existing REMOVEs.

4.3 View-change Agreement

We use the result of each process' CD proposal as input to a consensus protocol that drives agreement on a single view-change.

The consensus protocol in Rapid has a fast, leaderless path in the common case, that has the same overhead as simple gossip. The fast path is built around the Fast Paxos algorithm [53]. In our variation, we use the CD result as initial input to processes, instead of having an explicit proposer populating the processes with a proposal. Fast Paxos reaches a decision if there is a quorum larger than three quarters of the membership set with an identical proposal. Due to our prudent almost-everywhere CD scheme, with high probability, all processes indeed have an identical multi-process cut proposal. In this case, the VC protocol converges simply by counting the number of identical CD proposals.

The counting protocol itself uses gossip to disseminate and aggregate a bitmap of "votes" for each unique proposal. Each process sets a bit in the bitmap of a proposal to reflect its vote. As soon as a process has a proposal for which three quarters of the cluster has voted, it decides on that proposal.

If there is no fast-quorum support for any proposal because there are conflicting proposals, or a timeout is reached, Fast Paxos falls back to a recovery path, where we use classical Paxos [52] to make progress.

In the face of partitions [36], some applications may need to maintain availability everywhere (AP), and others only allow the majority component to remain live to provide strong consistency (CP). Rapid guarantees to reconfigure processes in the majority component. The remaining processes are forced to logically depart the system. They may wait to rejoin the majority component, or choose to form a separate configuration (which Rapid facilitates quickly). The history of the members forming a new configuration will have an explicit indication of these events, which applications can choose to use in any manner that fits them (including ignoring).

5 Logically Centralized Design

We now discuss how Rapid runs as a logically centralized service, where a set of auxiliary nodes S records the membership changes for a cluster $\mathscr{C}$. This is a similar model to how systems use ZooKeeper to manage membership: the centralized service is the ground truth of the membership list.

Only three minor modifications are required to the protocol discussed in §4:

1. Nodes in the current configuration $\mathscr{C}$ continue monitoring each other according to the k-ring topology (to scale the monitoring load). Instead of gossiping these alerts to all nodes in $\mathscr{C}$, they report it only to all nodes in S instead.
2. Nodes in S apply the CD protocol as before to identify a membership change proposal from the incoming alerts. However, they execute the VC protocol only among themselves.
3. Nodes in $\mathscr{C}$ learn about changes in the membership through notifications from S (or by probing nodes in S periodically).

The resulting solution inherits the stability and agreement properties of the decentralized protocol, but with reduced resiliency guarantees; the resiliency of the overall system is now bound to that of S ($F = \frac{S}{2} - 1$) – as with any logically centralized design. For progress, members of $\mathscr{C}$ need to be connected to a majority partition of S.

6 Implementation

Rapid is implemented in Java with 2362 lines of code (excluding comments and blank lines). This includes all the code associated with the membership protocol as well as messaging and failure detection. In addition, there are 2034 lines of code for tests. Our code is open-sourced under an Apache 2 license [2].

Our implementation uses gRPC and Netty for messaging. The counting step for consensus and the gossip-based dissemination of alerts are performed over UDP. Applications interact with Rapid using the APIs for joining and receiving callbacks described in §3. The logical identifier (§3) for each process is generated by the Rapid library using UUIDs. The join method allows users to supply edge failure detectors to use. Similar to APIs of existing systems such as Serf [44] and Akka Cluster [69], users associate application-specific metadata with the process being initialized (e.g., *"role":"backend"*).

Our default failure detector has observers send probes to their subjects and wait for a timeout. Observers mark an edge faulty when the number of communication exceptions they detect exceed a threshold (40% of the last 10 measurement attempts fail). Similar to Memberlist

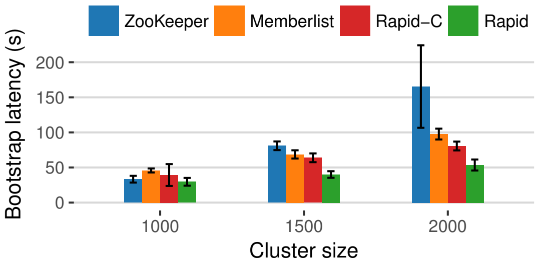


Figure 5: Bootstrap convergence measurements showing the time required for all nodes to report a cluster size of N. Rapid bootstraps a 2000 node cluster 2-2.32x faster than Memberlist, and 3.23-5.81x faster than ZooKeeper.

and Akka Cluster, Rapid batches multiple alerts into a single message before sending them on the wire.

7 Evaluation

Setup We run our experiments on a shared internal cloud service with 200 cores and 800 GB of RAM (100 VMs). We run multiple processes per VM, given that the workloads are not CPU bottlenecked. We vary the number of processes (N) in the cluster from 1000 to 2000.

We compare Rapid against *(i)* ZooKeeper [15] accessed using Apache Curator [13], *(ii)* Memberlist [47], the SWIM implementation used by Serf [44] and Consul [45]. For Rapid, we use the decentralized variant unless specified otherwise (Rapid-C, where a 3-node ensemble manages the membership of N processes).

We also tested Akka Cluster [69] but found its bootstrap process to not stabilize for clusters beyond 500 processes, and therefore do not present further (see §2.1 and Figure 1). All ZooKeeper experiments use a 3-node ensemble, configured according to [16]. For Memberlist, we use the provided configuration for single data center settings (called *DefaultLANConfig*). Rapid uses $\{K,H,L\} = \{10,9,3\}$ for all experiments and we also show a sensitivity analysis. We seek to answer:

- How quickly can Rapid bootstrap a cluster?
- How does Rapid react to different fault scenarios?
- How bandwidth intensive is Rapid?
- How sensitive is the almost-everywhere agreement property to the choice of K,H,L?
- Is Rapid easy to integrate with real applications?

Bootstrap experiments We stress the bootstrap protocols of all three systems under varying cluster sizes. For Memberlist and Rapid, we start each experiment with a single seed process, and after ten seconds, spawn a subsequent group of $N-1$ processes (for ZooKeeper, the 3-node ZooKeeper cluster is brought up first). Every process logs its observed cluster size every second. Every measurement is repeated five times per value of N. We measure the time taken for all processes to converge to a cluster size of N (Figure 5). For $N = 2000$, Rapid improves bootstrap latencies by 2-2.32x over Memberlist, and by 3.23-5.8x over ZooKeeper.

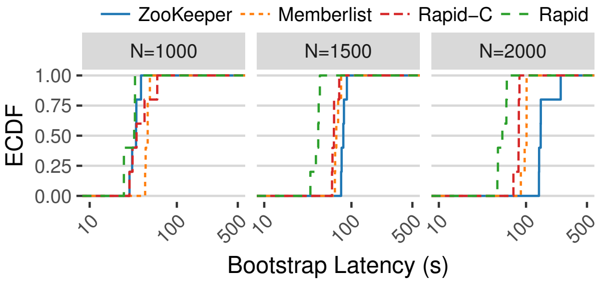


Figure 6: Bootstrap latency distribution for all systems.

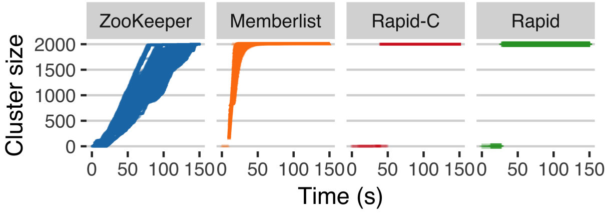


Figure 7: Timeseries showing the first 150 seconds of all three systems bootstrapping a 2000 node cluster.

ZooKeeper suffers from herd behavior during the bootstrap process (as documented in [18]), resulting in its bootstrap latency increasing by 4x from when N=1000 to when N=2000. Group membership with ZooKeeper is done using watches. When the i^{th} process joins the system, it triggers $i-1$ watch notifications, causing $i-1$ processes to re-read the full membership list and register a new watch each. In the interval between a watch having triggered and it being replaced, the client is not notified of updates, leading to clients observing different sequences of membership change events [17]. This behavior with watches leads to the eventually consistent client behavior in Figure 7. Lastly, we emphasize that this is a 3-node ZooKeeper cluster being used *exclusively* to manage membership for a single cluster. Adding even one extra watch per client to the group node at N=2000 inflates bootstrap latencies to 400s on average.

Memberlist processes bootstrap by contacting a seed. The seed thereby learns about every join attempt. However, non-seed processes need to periodically execute a push-pull handshake with each other to synchronize their views (by default, once every 30 seconds). Memberlist's convergence times are thereby as high as 95s on average when $N = 2000$ (Figure 7).

Similar to Memberlist, Rapid processes bootstrap by contacting a seed. The seed aggregates alerts until it bootstraps a cluster large enough to support a Paxos quorum (minimum of three processes). The remaining processes are admitted in a subsequent one or more view

System	N=1000	N=1500	N=2000
ZooKeeper	1000	1500	2000
Memberlist	901	1383	1858
Rapid-C	9	10	7
Rapid	4	8	4

Table 1: Number of unique cluster sizes reported by processes in bootstrapping experiments.

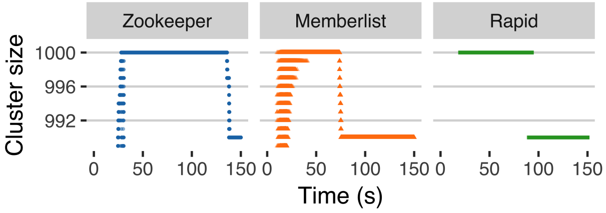


Figure 8: Experiment with 10 concurrent crash failures.

changes. For instance, in Figure 7, Rapid transitions from a single seed to a five node cluster, before forming a cluster of size 2000. We confirm this behavior across runs in Table 1, which shows the number of unique cluster sizes reported for different values of N. In the ZooKeeper and Memberlist experiments, processes report a range of cluster sizes between 1 and N as the cluster bootstraps. Rapid however brings up large clusters with very few intermediate view changes, reporting four and eight unique cluster sizes for each setting. Our logically centralized variant Rapid-C, behaves similarly for the bootstrap process. However, processes in Rapid-C periodically probe the 3-node ensemble for updates to the membership (the probing interval is set to be 5 seconds, the same as with ZooKeeper). This extra step increases bootstrap times over the decentralized variant; in the latter case, all processes participate in the dissemination of votes through aggregate gossip.

Crash faults We now set $N = 1000$ to compare the different systems in the face of crash faults. At this size, we have five processes per-core in the infrastructure, leading to a stable steady state for all three systems. We then fail ten processes and observe the cluster membership size reported by every other process in the system.

Figure 8 shows the cluster size timeseries as recorded by each process. Every dot in the timeseries represents a cluster size recording by a single process. With Memberlist and ZooKeeper, processes record several different cluster sizes when transitioning from N to $N - F$. Rapid on the other hand concurrently detects all ten process failures and removes them from the membership using a 1-step consensus decision. Note, our edge failure detector performs multiple measurements before announcing a fault for stability (§6), thereby reacting roughly 10 seconds later than Memberlist does. The results are identical when the ten processes are partitioned away completely from the cluster (we do not show the plots for brevity).

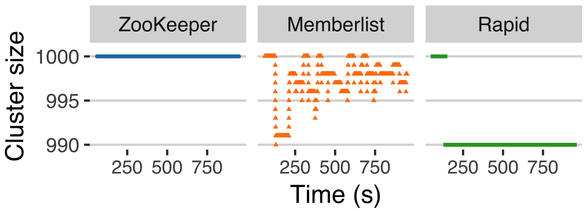


Figure 9: Asymmetric network failure with one-way network partition on the network interface of 1% of processes (ingress path).

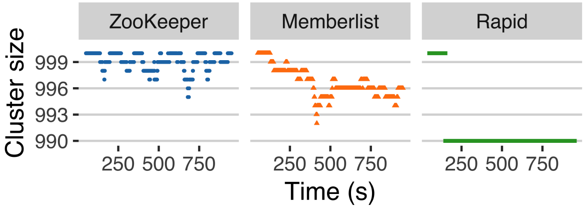


Figure 10: Experiment with 80% ingress packet loss on the network interface of 1% of processes.

Asymmetric network failures We study how each system responds to common network failures that we have seen in practice. These scenarios have also been described in [49, 19, 37, 67, 41].

Flip-flops in one-way connectivity. We enforce a "flip-flopping" asymmetric network failure. Here, 10 processes lose all packets that they receive for a 20 second interval, recover for 20 seconds, and then repeat the packet dropping. We enforce this by dropping packets in the `iptables INPUT` chain. The timeseries of cluster sizes reported by each process is shown in Figure 9.

ZooKeeper does not react to the injected failures because clients do not receive packets on the ingress path, but send heartbeats to the ZooKeeper nodes. Reversing the direction of connectivity loss as in the next experiment does cause ZooKeeper to react. Memberlist never removes all the faulty processes from the membership, and oscillates throughout the duration of the failure scenario. We also find several intervals of inconsistent views among processes. Unlike ZooKeeper and Memberlist, Rapid detects and removes the faulty processes.

High packet loss scenario. We now run an experiment where 80% of outgoing packets from the faulty processes are dropped. We inject the fault at $t = 90s$. Figure 10 shows the resulting membership size timeseries. ZooKeeper reacts to the failures at $t = 200s$, and does not remove all faulty processes from the membership. Figure 10 also shows how Memberlist's failure detector is conservative; even a scenario of sustained high packet loss is insufficient for Memberlist to conclusively remove a set of processes from the network. Further-

	KB/s (received / transmitted)		
System	Mean	p99	max
ZooKeeper	0.43 / 0.01	17.52 / 0.33	38.86 / 0.67
Memberlist	0.54 / 0.64	5.61 / 6.40	7.36 / 8.04
Rapid	0.71 / 0.71	3.66 / 3.72	9.56 / 11.37

Table 2: Mean, 99^{th} percentile and maximum network bandwidth utilization per process.

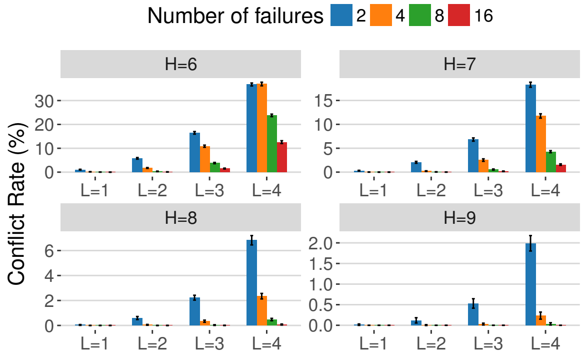


Figure 11: Almost-everywhere agreement conflict probability for different combinations of H, L and failures F when K=10. Note the different y-axis scales.

more, we observe view inconsistencies with Memberlist near $t = 400s$. Rapid, again, correctly identifies and removes only the faulty processes.

Memory utilization. Memberlist (written in Go) used an average of 12MB of resident memory per process. With Rapid and ZooKeeper agents (both Java based), GC events traced using `-XX:+PrintGC` report min/max heap utilization of 10/25MB and 3.5/16MB per process.

Network utilization. Table 2 shows the mean, 99th and 100th percentiles of network utilization per second across processes during the crash fault experiment (1000 processes). Rapid has a peak utilization of 9.56 KB/s received (and 11.37 KB/s transmitted) versus 7.36 KB/s received (8.04 KB/s transmitted) for Memberlist. Rapid therefore provides stronger guarantees than Memberlist for a similar degree of network bandwidth utilization. ZooKeeper clients have a peak ingress utilization of 38.86 KB/s per-process on average to learn the updated view of the membership.

K, H, L sensitivity study We now present the effect of K, H and L on the almost-everywhere agreement property of our multi-process detection technique. We initialize 1000 processes and select F random processes to fail. We generate alert messages from the F processes' observers and deliver these alerts to each process in a uniform random order. We count the number of processes that announce a membership proposal that did not include all F processes (a *conflict*). We run all parameter combinations for $H = \{6,7,8,9\}, L = \{1,2,3,4\}, F = \{2,4,8,16\}$ with 20 repetitions per combination.

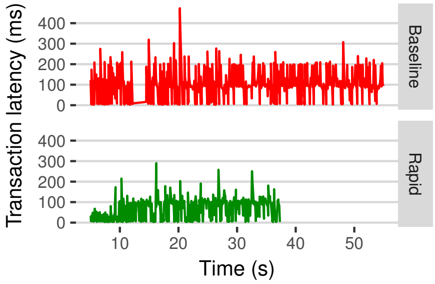


Figure 12: Transaction latency when testing an in-house gossip-style failure detector and Rapid for robustness against a communication fault between two processes. The baseline failure detector triggers repeated failovers that reduce throughput by 32%.

Figure 11 shows the results. As our analysis (§8) predicts, the conflict rate is highest when the gap between H and L is lowest ($H = 6, L = 4$) and the number of failures F is 2. This setting causes processes to arrive at a proposal without waiting long enough. As we increase the gap $H - L$ and increase F, the algorithm at each process waits long enough to gather all the relevant alerts, thereby diminishing the conflict probability. Our system is thereby robust across a range of values; for $H - L = 5$ and $F = 2$, we get a 2% conflict rate for different values of H and L. Increasing to $H - L = 6$ drops the probability of a conflict by a factor of 4.

Experience with end-to-end workloads We integrated Rapid within use cases at our organization that required membership services. Our goal is to understand the ease of integrating and using Rapid.

Distributed transactional data platform. We worked with a team that uses a distributed data platform that supports transactions. We replaced the use of its in-house gossip-based failure detector that uses all-to-all monitoring, with Rapid. The failure detector recommends membership changes to a Paxos-based reconfiguration mechanism, and we let Rapid provide input to the re-configuration management instead. Our integration added 62 and removed 25 lines of code. We also ported the system's failure detection logic such that it could be supplied to Rapid as an edge failure detector, which involved an additional 123 lines of code.

We now describe a use case in the context of this system where stable monitoring is required. For total ordering of requests, the platform has a transaction serialization server, similar to the one used in Google Megastore [20] and Apache Omid [14]. At any moment in time, the system has only one active serialization server, and its failure requires the cluster to identify a new candidate server for a failover. During this interval, workloads are paused and clients do not make progress.

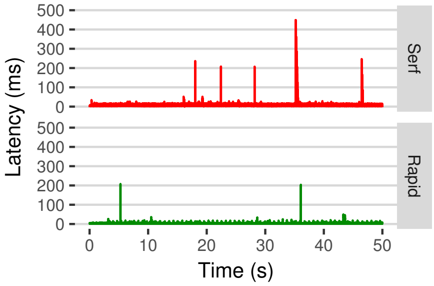


Figure 13: Service discovery experiment. Rapid's batching reduces the number of configuration reloads during a set of failures, thereby reducing tail latency.

We ran an experiment where two update-heavy clients (read-write ratio of 50-50) each submit 500K read/write operations, batched as 500 transactions. We injected a failure that drops all packets between the current serialization server and one other data server (resembling a packet blackhole as observed by [41]). Note, this fault does not affect communication links between clients and data servers. We measured the impact of this fault on the end-to-end latency and throughput.

With the baseline failure detector, the serialization server was repeatedly added and removed from the membership. The repeated failovers caused a degradation of end-to-end latency and a 32% drop in throughput (Figure 12). When using Rapid however, the system continued serving the workload without experiencing any interruption (because no node exceeded L reports).

Service discovery. A common use case for membership is service discovery, where a fleet of servers need to be discovered by dependent services. We worked with a team that uses Terraform [46] to drive deployments where a load balancer discovers a set of backend servers using Serf [44]. We replaced their use of Serf in this workflow with an agent that uses Rapid instead (the Rapid specific code amounted to under 20 lines of code). The setup uses nginx [1] to load balance requests to 50 web servers (also using nginx) that serve a static HTML page. All 51 machines run as t2.micro instances on Amazon EC2. Both membership services update nginx's configuration file with the list of backend servers on every membership change. We then use an HTTP workload generator to generate 1000 requests per-second. 30 seconds into the experiment, we fail ten nodes and observe the impact on the end-to-end latency (Figure 13).

Rapid detects all failures concurrently and triggers a single reconfiguration because of its multi-node membership change detection. Serf, which uses Memberlist, detects multiple independent failures that result in several updates to the nginx configuration file. The load balancer therefore incurs higher latencies when using Serf at multiple intervals (t=35s and t=46s) because nginx is reconfiguring itself. In the steady state where there are no failures, we observe no difference between Rapid and Serf, suggesting that Rapid is well suited for service discovery workloads, despite offering stronger guarantees.

8 Summary of Proofs

In the interest of space, we report the complete proof of correctness in a tech report [51], and only present the key take aways here. Our consensus engine is standard, and borrows from known literature on consensus algorithms [53, 32, 52]. We do not repeat its proof of Agreement and Liveness.

It is left to prove that faced with F failures in a configuration C, the stable failure detector detects and outputs F at all processes with high probability. We divide the proof into two parts.

Detection guarantee For parameters L and K, we can detect a failure of a set F as long as $|F|$ is bounded by the relationship $\frac{|F|}{|C|} \leq (1 - \frac{L}{K} - \frac{\lambda}{2K})$. Here, λ is the second eigenvalue of the underlying monitoring topology, and is tied to the expansion properties of the topology. In our experiments, with $K = 10$, we have observed consistently that $\frac{\lambda}{2K} < 0.45$ (which, for $L = 3$, yields $\frac{|F|}{|C|} \leq 0.25$).

Almost-everywhere agreement Second, we prove the almost-everywhere agreement property about our multi-process cut protocol. We assume that there are t failures, and that nodes receive alerts about these failures in a uniform random order. Let $\Pr[B(z)]$ be the probability that the CD protocol at z outputs a subset of t that differs from the output at other nodes. We show that if multiple processes fail simultaneously, $\Pr[B(z)]$ *exponentially* decreases with increasing K.

9 Conclusions

In this paper, we demonstrated the effectiveness of detecting cluster-wide multi-process cut conditions, as opposed to detecting individual node changes. The high fidelity of the CD output prevents frequent oscillations or incremental changes when faced with multiple failures. It achieves unanimous detection almost-everywhere, enabling a fast, leaderless consensus protocol to drive membership changes. Our implementation successfully bootstraps clusters of 2000 processes 2-5.8x times faster than existing solutions, while being stable against complex network failures. We found Rapid easy to integrate end-to-end within a distributed transactional data platform and a service discovery use case.

References

[1] Nginx. `http://nginx.org/en/docs/http/load_balancing.html`.

[2] Rapid source code. `https://github.com/lalithsuresh/rapid`, 2018.

[3] Adya, A., Myers, D., Howell, J., Elson, J., Meek, C., Khemani, V., Fulger, S., Gu, P., Bhuvanagiri, L., Hunter, J., Peon, R., Kai, L., Shraer, A., Merchant, A., and Lev-Ari, K. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 739–753.

[4] Amazon. Summary of the Amazon EC2, Amazon EBS, and Amazon RDS Service Event in the EU West Region. https://aws.amazon.com/message/2329B7/, 2011.

[5] Amazon. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. https://aws.amazon.com/message/65648/, 2011.

[6] Amazon. Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region. https://aws.amazon.com/message/680587/, 2012.

[7] Amazon. Summary of Windows Azure Service Disruption on Feb 29th, 2012. https://azure.microsoft.com/en-us/blog/summary-of-windows-azure\-service-disruption-on-feb-29th-2012/, 2012.

[8] Apache Cassandra. Apache Cassandra. http://cassandra.apache.org/, 2007.

[9] Apache Cassandra. Cassandra-3831: scaling to large clusters in GossipStage impossible due to calculatePendingRanges. https://issues.apache.org/jira/browse/CASSANDRA-3831, 2012.

[10] Apache Cassandra. Cassandra-6127: vnodes don't scale to hundreds of nodes. https://issues.apache.org/jira/browse/CASSANDRA-6127, 2013.

[11] Apache Cassandra. Cassandra-9667: strongly consistent membership and ownership. https://issues.apache.org/jira/browse/CASSANDRA-9667, 2015.

[12] Apache Cassandra. Cassandra-11740: Nodes have wrong membership view of the cluster. https://issues.apache.org/jira/browse/CASSANDRA-11740, 2016.

[13] Apache Curator. Apache Curator. https://curator.apache.org/, 2011.

[14] Apache Omid. Apache Omid. http://omid.incubator.apache.org/, 2011.

[15] Apache Zookeeper. Apache Zookeeper. https://zookeeper.apache.org/, 2010.

[16] Apache Zookeeper. ZooKeeper Administrator's Guide. https://zookeeper.apache.org/doc/trunk/zookeeperAdmin.html, 2010.

[17] Apache Zookeeper. ZooKeeper Programmer's Guide. https://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html, 2010.

[18] Apache Zookeeper. ZooKeeper Recipes. https://zookeeper.apache.org/doc/trunk/recipes.html, 2010.

[19] Bailis, P., and Kingsbury, K. The network is reliable. *Queue 12*, 7 (July 2014), 20:20–20:32.

[20] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.

[21] Barroso, L. A., Clidaras, J., and Hölzle, U. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2nd ed. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2013.

[22] Birman, K. Replication. Springer-Verlag, Berlin, Heidelberg, 2010, ch. A History of the Virtual Synchrony Replication Model, pp. 91–120.

[23] Burrows, M. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[24] Chockler, G. V., Keidar, I., and Vitenberg, R. Group communication specifications: A comprehensive study. *ACM Comput. Surv. 33*, 4 (Dec. 2001), 427–469.

[25] Consul Project. Issue 1212: Node health flapping - EC2. https://github.com/hashicorp/consul/issues/1212, 2015.

[26] Consul Project. Issue 916: Frequent membership loss for 50-100 node cluster in AWS. https://github.com/hashicorp/consul/issues/916, 2015.

[27] Consul Project. Issue 916: node health constantly flapping in large cluster. https://github.com/hashicorp/consul/issues/1337, 2015.

[28] Cowling, J., Ports, D. R. K., Liskov, B., Popa, R. A., and Gaikwad, A. Census: Location-aware membership management for large-scale distributed systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 12–12.

[29] Das, A., Gupta, I., and Motivala, A. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on* (2002), IEEE, pp. 303–312.

[30] Dean, J., and Barroso, L. A. The Tail At Scale. *Communications of the ACM 56* (2013), 74–80.

[31] Defago, X., Urban, P., Hayashibara, N., and Katayama, T. The ϕ accrual failure detector. In *RR IS-RR-2004-010, Japan Advanced Institute of Science and Technology* (2004), pp. 66–78.

[32] Dwork, C., Lynch, N., and Stockmeyer, L. Consensus in the presence of partial synchrony. *J. ACM 35*, 2 (Apr. 1988), 288–323.

[33] etcd. etcd. https://github.com/coreos/etcd, 2014.

[34] Friedman, J., Kahn, J., and Szemerédi, E. On the second eigenvalue of random regular graphs. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1989), STOC '89, ACM, pp. 587–598.

[35] Gafni, E., and Malkhi, D. Elastic Configuration Maintenance via a Parsimonious Speculating Snapshot Solution. In *DISC 2015* (Tokyo, France, Oct. 2015), Y. Moses and M. Roy, Eds., vol. LNCS 9363 of *29th International Symposium on Distributed Computing*, Toshimitsu Masuzawa and Koichi Wada, Springer-Verlag Berlin Heidelberg.

[36] Gilbert, S., and Lynch, N. Perspectives on the cap theorem. *Computer 45*, 2 (Feb. 2012), 30–36.

[37] Gill, P., Jain, N., and Nagappan, N. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 350–361.

[38] Goldreich, O. Studies in complexity and cryptography. Springer-Verlag, Berlin, Heidelberg, 2011, ch. Basic Facts About Expander Graphs, pp. 451–464.

[39] Gunawi, H. S., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Do, T., Adityatama, J., Eliazar, K. J., Laksono, A., Lukman, J. F., Martin, V., and Satria, A. D. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 7:1–7:14.

[40] Gunawi, H. S., Hao, M., Suminto, R. O., Laksono, A., Satria, A. D., Adityatama, J., and Eliazar, K. J. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 1–16.

[41] Guo, C., Yuan, L., Xiang, D., Dang, Y., Huang, R., Maltz, D., Liu, Z., Wang, V., Pang, B., Chen, H., Lin, Z.-W., and Kurien, V. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 139–152.

[42] Guo, Z., McDirmid, S., Yang, M., Zhuang, L., Zhang, P., Luo, Y., Bergan, T., Musuvathi, M., Zhang, Z., and Zhou, L. Failure recovery: When the cure is worse than the disease. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems* (Berkeley, CA, 2013), USENIX.

[43] Hadoop, A. http://hadoop.apache.org/.

[44] HashiCorp. Serf. https://www.serf.io/, 2013.

[45] HashiCorp. Consul. https://www.consul.io/, 2014.

[46] HashiCorp. Terraform. https://www.terraform.io/, 2014.

[47] HashiCorp. memberlist. https://github.com/hashicorp/memberlist, 2015.

[48] Huang, P., Guo, C., Zhou, L., Lorch, J. R., Dang, Y., Chintalapati, M., and Yao, R. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2017), HotOS '17, ACM, pp. 150–155.

[49] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems. http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf, 2009.

[50] Johansen, H. D., Renesse, R. V., Vigfusson, Y., and Johansen, D. Fireflies: A secure and scalable membership and gossip service. *ACM Trans. Comput. Syst. 33*, 2 (May 2015), 5:1–5:32.

[51] L. Suresh, D. Malkhi, P. Gopalan, I. Porto Carreiro, Z. Lokhandwala. Consistent and Stable Membership at Scale with Rapid. https://github.com/lalithsuresh/rapid/blob/master/docs/tech-report.pdf, 2018.

[52] Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[53] Lamport, L. Fast paxos. *Distributed Computing 19* (October 2006), 79–103.

[54] Leners, J. B., Wu, H., Hung, W.-L., Aguilera, M. K., and Walfish, M. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *SOSP* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 279–294.

[55] McCaffrey, C. Building Scalable Stateful Services. https://speakerdeck.com/caitiem20/building-scalable-stateful-services, 2015.

[56] Netflix. Dynomite. https://github.com/Netflix/dynomite, 2011.

[57] Netflix. Introducing Hystrix for Resilience Engineering. http://goo.gl/h9brP0, 2012.

[58] Netflix. Eureka. https://github.com/Netflix/eureka, 2014.

[59] Newell, A., Kliot, G., Menache, I., Gopalan, A., Akiyama, S., and Silberstein, M. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (April 2016), ACM - Association for Computing Machinery.

[60] Oki, B. M., and Liskov, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (1988), ACM, pp. 8–17.

[61] Peter Kelley. Eureka! Why You Shouldn't Use ZooKeeper for Service Discovery. https://tech.knewton.com/blog/2014/12/eureka-shouldnt-use\-zookeeper-service-discovery/, 2014.

[62] Redis. Redis. http://redis.io/, 2009.

[63] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[64] Scylla. ScyllaDB. http://www.scylladb.com/, 2013.

[65] Shraer, A., Reed, B., Malkhi, D., and Junqueira, F. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 39–39.

[66] Spiegelman, A., Keidar, I., and Malkhi, D. Dynamic reconfiguration: Abstraction and optimal asynchronous solution.

[67] Turner, D., Levchenko, K., Mogul, J. C., Savage, S., and Snoeren, A. C. On failure in managed enterprise networks. *HP Labs HPL-2012-101* (2012).

[68] Twitter. Finagle: A Protocol-Agnostic RPC System. https://goo.gl/ITebZs, 2011.

[69] Typesafe. Akka. http://akka.io/, 2009.

[70] Uber. Ringpop. https://ringpop.readthedocs.io/en/latest/index.html, 2015.

[71] Van Renesse, R., Birman, K. P., and Vogels, W. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst. 21*, 2 (May 2003), 164–206.

[72] van Renesse, R., Minsky, Y., and Hayden, M. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* (London, UK, UK, 1998), Middleware '98, Springer-Verlag, pp. 55–70.

On Smart Query Routing: For Distributed Graph Querying with Decoupled Storage

Arijit Khan
NTU Singapore

Gustavo Segovia
ETH Zurich, Switzerland

Donald Kossmann
Microsoft Research, Redmond, USA

Abstract

We study online graph queries that retrieve nearby nodes of a query node in a large network. To answer such queries with high throughput and low latency, we partition the graph and process in parallel across a cluster of servers. Existing distributed graph systems place each partition on a separate server, where query answering over that partition takes place. This design has two major disadvantages. First, the router maintains a fixed routing table (or, policy), thus less flexible for query routing, fault tolerance, and graph updates. Second, the graph must be partitioned so that the workload across servers is balanced, and the inter-machine communication is minimized. To maintain good-quality partitions, it is also required to update the existing partitions based on workload changes. However, graph partitioning, online monitoring of workloads, and dynamically updating the partitions are expensive.

We mitigate these problems by decoupling graph storage from query processors, and by developing smart routing strategies with graph landmarks and embedding. Since a query processor is no longer assigned any fixed part of the graph, it is equally capable of handling any request, thus facilitating load balancing and fault tolerance. Moreover, due to our smart routing strategies, query processors can effectively leverage their cache, reducing the impact of how the graph is partitioned across storage servers. Our experiments with several real-world, large graphs demonstrate that the proposed framework gRouting, even with simple hash partitioning, achieves up to an order of magnitude better query throughput compared to existing distributed graph systems that employ expensive graph partitioning and re-partitioning strategies.

1 INTRODUCTION

Graphs with millions of nodes and billions of edges are ubiquitous to represent highly interconnected structures including the World Wide Web, social networks, knowledge graphs, genome and scientific databases, medical and government records. To support online search and query services (possibly from many clients) with low latency and high throughput, data centers and cloud operators consider scale-out solutions, in which the graph and its data are partitioned horizontally across cheap commodity servers. We assume that the graph topology and the data associated with nodes and edges are co-located, since they are often accessed together [34, 16, 17]. Keeping with the modern database trends to support low-latency operations, we target a fully in-memory system, and use disks only for durability [35, 26, 28]. In this paper, we study online queries that explore a small region of the entire graph, and require fast response time. These queries usually start with a query node, and traverse its neighboring nodes up to a certain number of hops (we shall formally introduce our queries in Section 2). For efficiently answering online queries in a distributed environment, state-of-the-art systems (e.g., [35, 28, 26]) first partition the graph, and then place each partition on a separate server, where query answering over that partition takes place (Figure 1). Since the server which contains the query node can only handle that request, the router maintains a fixed routing table (or, a fixed routing strategy, e.g., modulo hashing). Hence, these systems are less flexible with respect to query routing and fault tol-

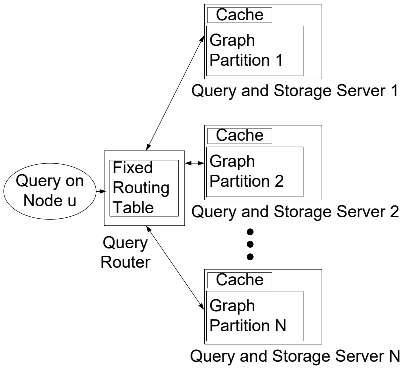


Figure 1: State-of-the-art distributed graph querying systems (e.g., SEDGE [35], Trinity [28], Horton [26])

erance, e.g., adding more machines will require updating the routing table. Besides, an effective graph partitioning in these systems must achieve: (1) workload balancing to maximize parallelism, and (2) locality of data access to minimize network communication. It has been demonstrated [35] that sophisticated partitioning schemes improve the performance of graph querying, compared to an inexpensive hash partitioning.

Due to power-law degree distribution of real-world graphs, it is difficult to get high-quality partitions [6]. Besides, a one-time partitioning cannot cope with later updates to graph structure or variations in query workloads. Several graph re- partitioning and replication-based strategies were proposed, e.g., [35, 18, 16]. However, online monitoring of workload changes, re-partitioning of the graph topology, and migration of graph data across servers are expensive; and they reduce the efficiency and throughput of online querying [25].

Our Contribution. In contrast to existing systems, we consider a different architecture, which relies *less* on an effective graph partitioning. Instead, we decouple query processing and graph storage into two separate tiers (Figure 2). In a decoupled framework, the graph is partitioned across servers allocated to the storage tier, and these storage servers hold the graph data in their main memory. Since a query processor is no longer assigned any fixed part of the graph, it is equally capable of handling any request, thus facilitating load balancing and fault tolerance. At the same time, the query router can send a request to any of the query processors, which adds more flexibility to query routing, e.g., more query processors can be added (or, a query processor that is down can be replaced) without affecting the routing strategy. Another benefit due to decoupled design is that each tier can be scaled-up independently. If a certain workload is processing intensive, more servers could be allocated to the processing tier. On the contrary, if the graph size increases over time, more servers can be added in the storage tier. This decoupled architecture, being generic, can be employed in many existing graph querying systems.

The idea of decoupling, though effective, is not novel. Facebook implemented a fast caching layer, Memcached on top of a graph database that scales the performance of graph query answering [19]. Google's F1 [29] and ScaleDB (http://scaledb.com/pdfs/TechnicalOverview.pdf) are based on a decoupling principle for scalability. Recently, Loesing et. al. [14] and Binnig et. al. [3] demonstrated the benefits of a decoupled, shared-data architecture, together with low latency and high throughput Infiniband network. Shalita et. al. [27] employed de-coupling for an optimal assignment of HTTP requests over a distributed graph storage.

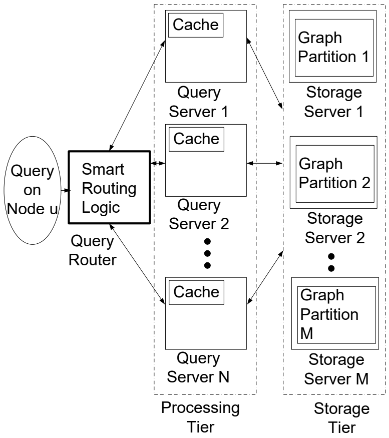


Figure 2: Decoupled architecture for graph querying

Our contribution lies in designing a smart query routing logic to utilize the cache of query processors over such decoupled architecture. Achieving more cache hits is critical in a decoupled architecture – otherwise, the query processors need to retrieve the data from storage servers, which will incur extra communication costs. This is a non-trivial problem, e.g., exploiting cache locality and balancing workloads are conflicting in nature. For example, to achieve maximum cache locality, the router can send all the queries to the same processor (assuming no cache eviction happens). However, the workload of the processors will be highly imbalanced, resulting in lower throughput. In addition, graph workloads are significantly different from traditional database applications. The interconnected nature of graph data results in poor locality, and each query usually accesses multiple neighboring nodes spreading across the distributed storage. Therefore, to maximize cache hit rates at query processors, it is not sufficient to only route the queries on same nodes to the same processor. Rather, successive queries on *neighboring* nodes should also be routed to the same processor, since the neighborhoods of two nearby nodes may significantly overlap. To the best of our knowledge, such smart query routing schemes for effectively leveraging the cache contents were not considered in existing graph querying systems.

We summarize our contributions as follows.

1. We study for the first time the problem of *smart query routing* aimed at improving the throughput and efficiency of distributed graph querying.
2. In contrast to many distributed graph querying systems [35, 28, 26], we consider a different architecture that *decouples* query processors from storage layer, thereby achieving flexibility in system deployment, query routing, scaling up, load balancing, and fault tolerance.
3. We develop smart, lightweight, and adaptive *query routing algorithms* that improve cache hit rates at query processors, thus reducing communication

with storage layer, and making our design less reliant on a sophisticated graph partitioning scheme across storage layer.

4. We empirically demonstrate throughput and efficiency of our framework, gRouting on three real-life graphs, while also comparing with two existing distributed graph processing systems (SEDGE/Giraph [35] and PowerGraph [6]). Our decoupled implementation, even with its simple hash partitioning, achieves up to an order of magnitude higher throughput compared to existing systems with expensive graph partitioning schemes.

2 PRELIMINARIES

2.1 Graph Data Model

A heterogeneous network can be modeled as a labeled, directed graph $G = (V, E, \mathscr{L})$ with node set V, edge set E, and label set $\mathscr{L}$, where (1) each node $u \in V$ represents an entity in the network, (2) each directed edge $e \in E$ denotes the relationship between two entities, and (3) $\mathscr{L}$ is a function which assigns to each node u and every edge e a label $\mathscr{L}(u)$ (and $\mathscr{L}(e)$, respectively) from a finite alphabet. The node labels represent the attributes of the entities, e.g., name, job, location, etc, and edge labels the type of relationships, e.g., founder, place founded, etc.

We store the graph as an adjacency list. Every node in the graph is added as an entry in the storage where the key is the node id and the value is an array of 1-hop neighbors. If the nodes and edges have labels, they are stored in the corresponding value entry. For each node, we store both incoming and outgoing edges. Both incoming and outgoing edges of a node can be important from the context of different queries. As an example, if there is an edge *founded* from *Jerry Yang* to *Yahoo!* in a knowledge graph, there also exists a reverse relation *founded_by* from *Yahoo!* to *Jerry Yang*. Such information could be useful in answering queries about *Yahoo!*.

2.2 h-Hop Traversal Queries

We discuss various h-hop queries over heterogeneous, directed graphs in the following.

1. **h-hop Neighbor Aggregation**: Count the number of h-hop neighbors of a query node.
2. **h-step Random Walk with Restart**: The query starts at a node, and runs for h-steps — at each step, jumps to one of its neighbors with equal probability, or returns to the query node with a small probability.
3. **h-hop Reachability**: Find if a given target node is reachable from a given source node within h-hops.

The aforementioned queries are often used as the basis for more complex graph operations. For example, neighborhood aggregation is critical for node labeling and classification, that is, the label of an unlabeled node could be assigned as the most frequent label which is present within its h-hop neighborhood. The h-step random walk is useful in expert finding, ranking, discovering functional modules, complexes, and pathways. Our third query can be employed in distance-constrained and label-constrained reachability search, as well as in approximate graph pattern matching queries [16].

2.3 Decoupled Design

We decouple query processing from graph storage. This decoupling happens at a *logical* level. As an example, query processors can be different physical machines than storage servers. On the other hand, the same physical machine can also run a query processing daemon, together with storing a graph partition in its main memory as a storage server. However, the logical separation between the two layers is critical in our design.

The advantages of this separation are more flexibility in query routing, system deployment, and scaling up, as well as achieving better load balancing and fault tolerance. However, we must also consider the drawbacks of having the graph storage apart. First, query processors may need to communicate with the storage tier via the network. This includes an additional penalty to the response time for answering a query. Second, it is possible that this design causes high contention rates on either the network, storage tier, or both.

To mitigate these issues, we design smart routing schemes that route queries to processors which are likely to have the relevant data in their cache, thereby reducing the communication overhead between processing and storage tiers. Below, we discuss various components of our design, including storage, processing tier, and router.

Graph Storage Tier. The storage tier holds all graph data by horizontally partitioning it across cheap commodity servers. Sophisticated graph partitioning will benefit our decoupled architecture as follows. Let us assume that the neighboring nodes can be stored in a page within the same storage server, and the granularity of transfer from storage to processing tier is a page containing several nodes. Then, we could actually ship a set of relevant nodes with a single request if the graph is partitioned well. This will reduce the number of times data are transferred between the processing and storage tier.

However, our lightweight and smart query routing techniques exploit the notion of graph landmarks [12] and embedding [36], thereby effectively utilizing the cache of query processors that stores recently used graph data. As demonstrated in our experiments, due to our smart routing, many neighbors up to 2∼3-hops of a query node can be found locally in the query processors' cache. Therefore, the partitioning scheme employed across storage servers becomes less important.

Query Processing Tier. The processing tier consists of servers where the actual query processing takes place. These servers do not communicate with each other [14]. They only receive queries from the query router, and also request graph data from the storage tier if necessary.

To reduce the amount of calls made to the storage tier, we utilize the cache of the query processors. Whenever some data is retrieved from the storage, it is saved in cache, so that the same request can be avoided in the near future. However, it imposes a constraint on the maximum storage capacity. When the addition of a new entry surpasses this storage limit, one or more old entries are evicted from the cache. We select the LRU (i.e., Least Recently Used) eviction policy because of its simplicity. LRU is usually implemented as the default cache replacement policy, and it favors recent queries. Thus, it performs well with our smart routing schemes.

Query Router. The router creates a thread for each processor, and opens a connection to send queries by following the routing schemes which we shall describe next.

3 QUERY ROUTING STRATEGIES

When a query arrives at the router, the router decides the appropriate query processor to which the request could be sent. For existing graph querying systems, e.g., SEDGE [35] and Horton [26], where each query processor is assigned a graph partition, this decision is fixed and defined in the routing table; the processor which contains the query node handles the request. With a decoupled architecture, no such mapping exists. Hence, we design novel routing schemes with the following objectives.

3.1 Routing Algorithm Objectives

1. Leverage each processor's cached data. Let us consider t successive queries received by the router. The router will send them to query processors in a way such that the average number of *cache hits* at the processors is maximized. This, in turn, reduces the average query processing latency. However, as stated earlier, to achieve maximum cache hits, it will not be sufficient to only route the queries on same nodes to the same processor. Rather, successive queries on *neighboring* nodes should also be routed to the same processor, since the neighborhoods of two nearby nodes may significantly overlap. This will be discussed shortly in Requirement 1.

2. Balance workload even if skewed or contains hotspot. As earlier, let us consider a set of t successive queries. A naïve approach will be to ensure that each query processor receives equal number of queries, e.g., a round-robin way of query dispatching by the router. However, each query might have a different workload, and would require a different processing time. We, therefore, aim at maximizing the overall *throughput* via query stealing (explained in Requirement 2), which automatically balances the workload across query processors.

3. Make fast routing decisions. The average time at the router to dispatch a query should be minimized, ideally a small constant time, or much smaller than $\mathcal{O}(n)$, where n is the number of nodes in the input graph. This reduces the query processing latency.

4. Have low storage overhead in the router. The router may store auxiliary data to enable fast routing decisions. However, this additional storage overhead must be a small fraction compared to the graph size.

3.2 Challenges in Query Routing

It is important to note that our routing objectives are not in harmony; in fact, they are often conflicting with each other. First, in order to achieve maximum cache locality, the router can send all the queries to the same processor (assuming no cache eviction happens). However, the workload of the processors will be highly imbalanced in this case, resulting in lower throughput. Second, the router could inspect the cache of each processor before making a good routing decision, but this will add network communication delay. Hence, the router must *infer* what is likely to be in each processor's cache.

In the following, we introduce two concepts that are directly related to our routing objectives, and will be useful in designing smart routing algorithms.

Topology-Aware Locality. To understand the notion of cache locality for graph queries (i.e., routing objective 1), we define a concept called *topology-aware locality*. If u and v are nearby nodes, then successive queries on u and v must be sent to the same processor. It is very likely that the h-hop neighborhoods of u and v significantly overlap.

But, how will the router know that u and v are nearby nodes? One option is to store the entire graph topology in the router; but this could have a high storage overhead. For example, the *WebGraph* dataset that we experimented with has a topology of size 60GB. Ideally, a graph with 10^7 nodes can have up to 10^{14} edges, and in such cases, storing only the topology itself requires petabytes of memory. Thus, we impose a requirement on our smart routing schemes as follows.

Requirement 1 *The additional storage at the router for enabling smart routing should not be asymptotically larger than $\mathcal{O}(n)$, n being the number of nodes; however, the routing schemes should still be able to exploit topology-aware locality.*

Achieving this goal is non-trivial, as the topology size can be $\mathcal{O}(n^2)$, and we provision for only $\mathcal{O}(n)$ space to approximately preserve such information.

Query Stealing. Routing queries to processors that have the most useful cache data might not always be the best strategy. Due to power-law degree distribution of real-world graphs, processing queries on different nodes might require different amount of time. Therefore, the processors dealing with high-degree nodes will have more workloads. Load imbalance can also happen if queries are concentrated in one specific region of the graph. When that happens, all queries will be sent to one processor, while other processors remain idle. To rectify such scenarios, we implement *query stealing* in our routing schemes as stated next.

Requirement 2 *Whenever a processor is idle and is ready to handle a new query, if it does not have any other requests assigned to it, it may "steal" a request that was originally intended for another processor.*

Query stealing is a well established technique for load balancing that is prevalently used by the HPC community, and there are several ways how one can implement it. *We perform query stealing at the router level.* In particular, the router sends the next query to a processor only when it receives an acknowledgement for the previous query from that processor. The router also keeps a *queue* for each connection in order to store the future queries that need to be delivered to the corresponding processor. By monitoring the length of these queues, it can estimate how busy a processor is, and this enables the router to rearrange the future queries for load balancing. We demonstrate the effectiveness of query stealing in our experiments (Section 4.6).

We next design four routing schemes — the first two are naïve and do not meet all the objectives of smart routing. On the other hand, the last two algorithms follow the requirements of a smart routing strategy.

3.3 Baseline Methods

3.3.1 Next Ready Routing

Next Ready routing is our first baseline strategy. The router decides where to send a query by choosing the next processor that has finished computing and is ready for a new request. The main advantages are: (1) It is easy to implement. (2) Routing decisions are made in constant time. (3) No preprocessing or storage overhead is required. (4) The workload is well balanced. However, this scheme fails to leverage processors' cache.

3.3.2 Hash Routing

The second routing scheme that we implement is *hash*, and it also serves as a baseline to compare against our smart routing techniques. The router applies a fixed hash function on each query node's id to determine the processor where it sends the request. In our implementation, we apply a modulo hash function.

In order to facilitate load balancing in the presence of workload skew, we implement *query stealing* mechanism. Whenever a processor is idle and is ready to handle a new query, if it does not have any other requests assigned to it, it steals a request that was originally intended for another processor. Since queries are queued in the router, the router is able to take this decision, and ensures that there are no idle processors when there is still some work to be done. Our hash routing has all the benefits of next ready, and very likely it sends a repeated query to the same processor, thereby getting better locality out of the cache. However, hash routing cannot capture topology-aware locality.

3.4 Proposed Methods

3.4.1 Landmark Routing

Our first smart routing scheme is based on landmark nodes [12]. One may recall that we store both incoming and outgoing edges of every node, thus we consider a *bi-directed* version of the input graph in our smart routing algorithms. We select a small set L of nodes as landmarks, and also pre-compute the distance of every node to these landmarks. We determine the optimal number of landmarks based on empirical results. Given some landmark node $l \in L$, the distance $d(u,v)$ between any two nodes u and v are bounded as follows:

$$|d(u,l) - d(l,v)| \leq d(u,v) \leq d(u,l) + d(l,v) \qquad (1)$$

Intuitively, if two nodes are close to a given landmark, they are likely to be close themselves. Our landmark routing is based on the above principle. We first select a set of landmarks that partitions the graph into P regions, where P is the total number of processors. We then decide a one-to-one mapping between those regions and processors. Now, if a query belongs to a specific region (decided based on its distance to landmarks), it is routed to the corresponding processor. Clearly, this routing strategy requires a preprocessing phase as follows.

Preprocessing. We select landmarks based on their node degree and how well they spread over the graph [1]. Our first step is to find a certain number of landmarks considering the highest degree nodes, and then compute their distance to every node in the graph by performing breadth first searches (BFS). If we find two landmarks to be closer than a pre-defined threshold, the one with the lower degree is discarded. The complexity of this step is $\mathcal{O}(|L|e)$, due to $|L|$ number of BFS, where $|L|$ is the number of landmarks, and e is the number of edges.

Next, we assign the landmarks to query processors as follows. First, every processor is assigned a "pivot" landmark with the intent that pivot landmarks are as far from each other as possible. The first two pivot landmarks are the two that are farthest apart considering all other landmark pairs. Each next pivot is selected as the landmark that is farthest from all previously selected pivot landmarks. Each remaining landmark is assigned to the processor which contains its closest pivot landmark. The complexity of this step is $\mathcal{O}(|L|^2 + |L|P)$, where P is the number of processors.

Finally, we define a "distance" metric d between the graph nodes and query processors. The distance of a node u to a processor p is defined as the minimum distance of u to any landmark that is assigned to processor p. This information is stored in the router, which requires $\mathscr{O}(nP)$ space and $\mathscr{O}(nL)$ time to compute, where n is the number of nodes. Therefore, *the storage requirement at the router is linear in the number of nodes*.

Routing. To decide where to send a query on node u, the router verifies the pre-computed distance $d(u,p)$ for every processor p, and selects the one with the smallest $d(u,p)$ value. As a consequence, the routing decision time is linear in the number of processors: $\mathscr{O}(P)$. This is very efficient since the number of processors is small.

In contrast to our earlier baseline routings, this method is able to leverage topology-aware locality. It is likely that query nodes that are in the neighborhood of each other will have similar distances to the processors; hence, they will be routed in a similar fashion. On the other hand, the landmark routing scheme is less flexible with respect to addition or removal of processors, since the assignment of landmarks to processors, as well as the distances $d(u,p)$ for every node u and each processor p needs to be recomputed.

The distance metric $d(u,p)$ is useful not only in finding the best processor for a certain query, but it can also be used for load balancing, fault tolerance, dealing with workload skew, and hotspots. As an example, let us assume that the closest processor for a certain query is very busy, or is currently down. Since the distance metric gives us distances to all processors, the router is able to select the second, third, or so on closest processor. This form of load balancing will impact the nearby query nodes in the same way; and therefore, the modified routing scheme will still be able to capture topology-aware locality. In practice, it can be complex to define exactly when a query should be routed to its next best query processor. We propose a formula that calculates the *load-balanced distance* $d^{LB}(u,p)$ as given below.

$$d^{LB}(u,p) = d(u,p) + \frac{\text{Processor Load}}{\text{Load Factor}} \tag{2}$$

Thus, the query is always routed to the processor with the smallest $d^{LB}(u,p)$. The router uses the number of queries in the queue corresponding to a processor as the measure of its load. The load factor is a tunable parameter, which allows us to decide how much load would result in the query to be routed to another processor. We find its optimal value empirically.

Dealing with Graph Updates. During addition/ deletion of nodes and edges, one needs to recompute the distances from every node to each of the landmarks. This can be performed efficiently by keeping an additional *shortest-path-tree* data structure [31]. However, to avoid the additional space and time complexity of maintaining a shortest-path-tree, we follow a simpler approach. When a new node u is added, we compute the distance of this node to every landmark, and also its distance $d(u,p)$ to every processor p. In case of an edge addition or deletion between two existing nodes, for these two end-nodes and their neighbors up to a certain number of hops (e.g., 2-hops), we recompute their distances to every landmark, as well as to every processor. Finally, in case of a node deletion, we handle it by considering deletion of multiple edges that are incident on it. After a significant number of updates, previously selected landmark nodes become less effective; thus, we recompute the entire preprocessing step periodically in an off-line manner.

3.4.2 Embed Routing

Our second smart routing scheme is the *Embed* routing, which is based on graph embedding [36, 4]. We embed a graph into a lower dimensional Euclidean space such that the hop-count distance between graph nodes are approximately preserved via their Euclidean distance (Figure 3). We then use the resulting node co-ordinates to determine how far a query node is from the recent history of queries that were sent to a specific processor. Clearly, embed routing also requires a preprocessing step.

Preprocessing. For efficiently embedding a large graph in a D-dimensional Euclidean plane, we first select a set L of landmarks and find their distances from each node in the graph. We then assign co-ordinates to landmark nodes such that the distance between each pair of landmarks is approximately preserved. We, in fact, minimize the *relative error* in distance for each pair of landmarks, defined below.

$$f_{error}(v_1,v_2) = \frac{|d(v_1,v_2) - \text{EuclideanDist}(v_1,v_2)|}{d(v_1,v_2)} \tag{3}$$

Here, $d(v_1,v_2)$ is the hop-count distance between v_1 and v_2 in the original graph, and EuclideanDist(v_1,v_2) is their Euclidean distance after the graph is embedded. We minimize the relative error since we are more interested in preserving the distances between nearby node pairs. Our problem is to minimize the aggregate of such errors over all landmark pairs — this can be cast as a generic multi-dimensional global minimization problem, and could be approximately solved by many off-the-shelf techniques, e.g., the *Simplex Downhill* algorithm that we apply in this work. Next, every other node's co-ordinates are found also by applying the Simplex Downhill algorithm that minimizes the aggregate relative distance error between the node and all the landmarks. The overall graph embedding procedure consumes a modest preprocessing time: $\mathscr{O}(|L|e)$ due to BFS from $|L|$ landmarks, $\mathscr{O}(|L|^2D)$ for embedding the landmarks, and $\mathscr{O}(n|L|D)$ for embedding the remaining nodes. In addition, the second step is completely parallelizable per node. Since each node receives D co-ordinates, it requires total $\mathscr{O}(nD)$ space in the router, which is linear in the number of nodes. Unlike landmark routing, *a benefit*

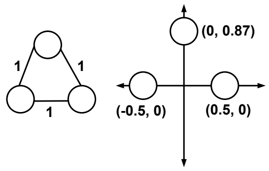


Figure 3: Example of graph embedding in 2D Euclidean plane

of embed routing is that the preprocessing is independent of the system topology, allowing more processors to be easily added at a later time.

Routing. The router has access to each node's co-ordinates. By keeping an average of the query nodes' co-ordinates that it sent to each processor, it is able to infer the cache contents in these processors. As a consequence, the router finds the distance between a query node u and a processor p, denoted as $d_1(u,p)$, and defined as the distance of the query node's co-ordinates to the historical mean of the processor's cache contents. As recent queries are more likely to influence the cache contents due to LRU eviction policy, we use the exponential moving average to compute the mean of the processor's cache contents. Initially, the mean co-ordinates for each processor are assigned uniformly at random. Next, assuming that the last query on node v was sent to processor p, its updated mean co-ordinates are:

$$\text{MeanCo-ordinates}(p) = \alpha \cdot \text{MeanCo-ordinates}(p) + (1-\alpha) \cdot \text{Co-ordinates}(v) \quad (4)$$

The smoothing parameter $\alpha \in (0,1)$ in the above Equation determines the degree of decay used to discard older queries. For example, α close to 0 assigns more weight only to the last query, and α close to 1 decreases the weight on the last query. We determine the optimal value of α based on experimental results. Finally, the distance between a query node u and a processor p is computed as given below.

$$d_1(u,p) = ||\text{MeanCo-ordinates}(p) - \text{Co-ordinates}(u)|| \quad (5)$$

Since we embed in an Euclidean plane, we use the L_2 norm to compute distances. We select the processor with the smallest $d_1(u,p)$ distance. One may observe that the routing decision time is only $\mathcal{O}(PD)$, P being the number of processors and D the number of dimensions.

Analogous to landmark routing, we now have a distance to each processor for a query; and hence, we are able to make routing decisions taking into account the processors' workloads and faults. As earlier, we define a *load-balanced distance* $d_1^{LB}(u,p)$ between a query node u and a processor p, and the query is always routed to the processor with the smallest $d_1^{LB}(u,p)$ value.

$$d_1^{LB}(u,p) = d_1(u,p) + \frac{\text{Processor Load}}{\text{Load Factor}} \quad (6)$$

The embed routing has all the benefits of smart routing. This routing scheme divides the active regions (based on workloads) of the graph into P partitions in an overlapping manner, and assigns them to the processors' cache. Moreover, it dynamically adapts the partitions with new workloads. Therefore, *it bypasses the expensive graph partitioning and re-partitioning problems to the existing cache replacement policy of the query processors*. This shows the effectiveness of embed routing.

Dataset	# Nodes	# Edges	Size on Disk (Adj. List)
WebGraph	105 896 555	3 738 733 648	60.3 GB
Memetracker	96 608 034	418 237 269	8.2 GB
Freebase	49 731 389	46 708 421	1.3 GB

Table 1: Graph datasets

Dealing with Graph Updates. Due to pre-assignment of node co-ordinates, embed routing is less flexible with respect to graph updates. When a new node is added, we compute its distance from the landmarks, and then assign co-ordinates to the node by applying the Simplex Downhill algorithm. Edge updates and node deletions are handled in a similar method as discussed for landmark routing. We recompute the entire preprocessing step periodically in an off-line manner to deal with a significant number of graph updates.

4 EVALUATION

4.1 Experiment Setup

• **Cluster Configuration.** We perform experiments on a cluster of 12 servers having 2.4 GHz Intel Xeon processors, and interconnected by 40 Gbps Infiniband, and also by 10 Gbps Ethernet. Most experiments use a single core of each server with the following configuration: 1 server as router, 7 servers in the processing tier, 4 servers in the storage tier; and communication over Infiniband with remote direct memory access (RDMA). Infiniband allows RDMA in a few microseconds. We use a limited main memory (0∼4GB) as the cache of processors. Our codes are implemented in C++.

To implement our storage tier, we use RAMCloud [20], which provides high throughput and very low read/write latency, in the order of 5-10 μs for every put/ get operation. It is able to achieve this efficiency because it keeps all stored values in memory as a distributed key-value store, where a key is hashed to determine on which server the corresponding key-value pair will be stored.

• **Datasets.** We summarize our data sets in Table 1. As explained in Section 2, we store both in- and out-neighbors. The graph is stored as an adjacency list — every node-id in the graph is the key, and the corresponding value is an array of its 1-hop neighbors. The graph is partitioned across storage servers via RAMCloud's default and inexpensive hash partitioning scheme, MurmurHash3 over graph nodes.

WebGraph: The uk-2007-05 web graph (http://law.di.unimi.it/ datasets.php) is a collection of web pages, which are represented as nodes, and their hyperlinks as edges. **Memetracker:** This dataset (snap.stanford.edu) tracks quotes and phrases that appeared from August 1 to October 31, 2008 across online news spectrum. We consider documents as nodes and hyper-links as edges. **Freebase:** We download the *Freebase* knowledge graph from http://www.freebase.com/. Nodes are named entities (e.g., Google) or abstract concepts (e.g., Asian people), and edges denote relations (e.g., founder).

• **Online Query Workloads.** We consider three online graph queries [35], discussed in Section 2.2 — all require traversals up to h hops: (1) h-hop neighbor aggregation, (2) h-step random walk with restart, and (3) h-hop reachability. We consider a uniform mixture of above queries. We simulate a scenario when queries are drawn from a hotspot region; and the hotspots change over time. In particular, we select 100 nodes from the graph uniformly at random. Then, for each of these nodes, we select 10 different query nodes which are at most r-hops away from that node. Thus, we generate 1000 queries; every 10 of them are from one hotspot region, and the pairwise distance between any two nodes from the same hotspot is at most $2r$. Finally, all queries from the same hotspot are grouped together and sent consecutively. We report our results averaged over 1000 queries.

To realize the effect of topology-aware locality, we consider smaller values of r and h, e.g., $r = 2$ and $h = 2$.

• **Evaluation Metrics.**
Query Response Time measures the average time required to answer one query.
Query Processing Throughput measures the number of queries that can be processed per unit time.
Cache Hit Rate: We report cache hit rates, since higher cache hit rates reduce the query response time. Consider t queries $q_1, q_2, \ldots, q_t$ received successively by the router. For simplicity, let us assume that each query retrieves all h-hop neighbors of that query node (i.e., h-hop neighborhood aggregation). We denote by $|N_h(q_i)|$ the number of nodes within h-hops from q_i. Among them, we assume that $|N_h^c(q_i)|$ number of nodes are found in the query processors' cache.

$$\text{Cache Hit Rates} := \sum_{i=1}^{t} |N_h^c(q_i)| \tag{7}$$

$$\text{Cache Miss Rates} := \sum_{i=1}^{t} \left(|N_h(q_i)| - |N_h^c(q_i)|\right) \tag{8}$$

• **Parameter Setting.** We find that *embed routing performs the best compared to three other routing strategies*. We also set the following parameter values since they perform the best in our implementation. We shall, however, demonstrate sensitivity of our routing algorithms with these parameters in Section 4.6.

We use maximum 4GB cache in each query processor. All experiments are performed with the cache initially empty (cold cache). The number of landmarks $|L|$ is set as 96 with at least 3 hops of separation from each other. For graph embedding, 10 dimensions are used. Load Factor (which impacts query stealing) is set as 20, and the smoothing parameter $\alpha = 0.5$.

In order to realize how our routing schemes perform when there is no cache in processors, we consider an additional "no-cache" scheme. In this mode, all queries are routed following the next ready technique; however, as there is no cache in query processors, there will be no overhead due to cache lookup and maintenance.

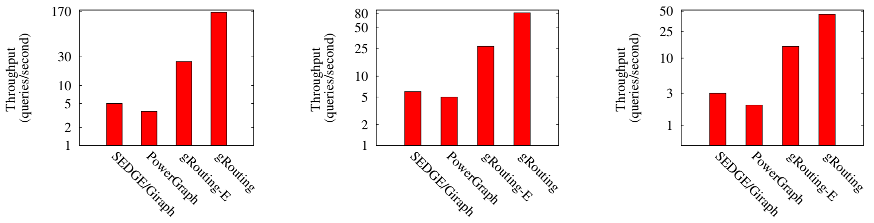
• **Compared Systems.** Decoupled architecture and our smart routing logic, being generic, can benefit many graph querying systems. Nevertheless, we compare gRouting with two distributed graph processing systems: SEDGE/Giraph [35] and PowerGraph [6]. Other recent graph querying systems, e.g., [26, 19] are not publicly available for a direct comparison.

SEDGE [35] was developed for h-hop traversal queries on top of Giraph or Google's Pregel system [15]. It follows in-memory, vertex-centric, bulk-synchronous parallel model. SEDGE employs ParMETIS software [9] for graph partitioning and re-partitioning. PowerGraph [6] follows in-memory, vertex-centric, asynchronous gather-apply-scatter model. In the beginning, only the query node is active, and each active node then activates its neighbors, until all the h-hop neighbors from the query nodes are activated. PowerGraph also employs a sophisticated node-cut based graph partitioning method.

4.2 Comparison with Graph Systems

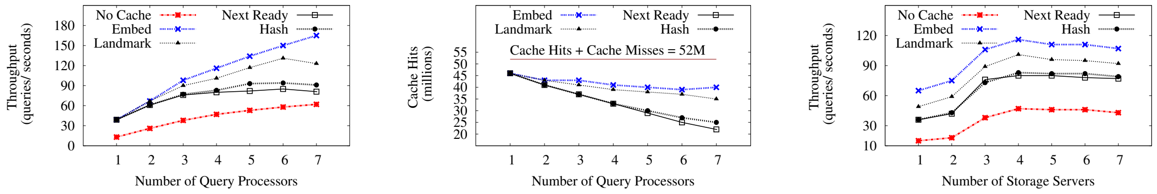
We compare gRouting (embed routing is used) with two distributed graph processing systems, SEDGE/Giraph [35] and PowerGraph [6]. As these systems run on Ethernet, we consider a version of gRouting on Ethernet (gRouting-E). We consider 12 machines configuration of SEDGE and PowerGraph, since query processing and graph storage in them are coupled on same machines. In contrast, we fix the number of routing, processing, and storage servers as 1, 7 and 4, respectively. The average 2-hop neighborhood size varies from 10K∼60K nodes over our datasets.

In Figure 4, we find that our throughput, with hash partitioning and over Ethernet, is 5∼10 times better than SEDGE and PowerGraph that employ expensive graph partitioning and re- partitioning. The re-partitioning in SEDGE requires around 1 hour and also apriori information on future queries, whereas PowerGraph graph partitioning finishes in 30 min. On the contrary, gRouting performs lightweight hash partitioning over graph nodes, and does not require any prior knowledge of the future workloads. Moreover, our throughput over Infiniband is 10∼35 higher than these systems. *These results show the usefulness of smart query routing over expensive graph partitioning and re-partitioning schemes.*



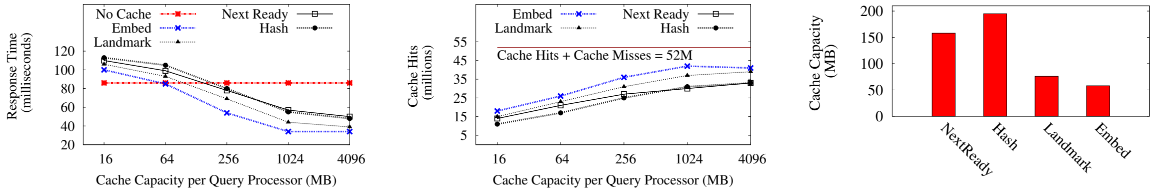
(a) *WebGraph* (b) *MemeTracker* (c) *Freebase*

Figure 4: Throughput comparison



(a) Throughput (b) Cache Hits (c) Throughput

Figure 5: Performance with varying number of query processors and storage servers, *WebGraph*



(a) Response time (b) Cache hits (c) Min. cache required to reach No-Cache's response time

Figure 6: Impact of cache size, *WebGraph*

Next, we report scalability, impact of cache sizes, and graph updates over our largest *Webgraph* dataset, and using Infiniband network.

4.3 Scalability and Deployment Flexibility

One of the main benefits of separating processing and storage tiers is deployment flexibility — they can be scaled-up independently, which we investigate below.

Processing Tier: We vary the number of processing servers from 1 to 7, while using 1 router and 4 storage servers. In Figure 5(a), we show throughput with varying number of processing servers. Corresponding cache hit rates are presented in Figure 5(b). For these experiments, we assume that each query processor has sufficient cache capacity (4GB) to store the results of all 1000 queries (i.e, adjacency lists of 52M nodes, shown in Figure 5(b)). Since, for every experiment, we start with an empty cache, and then send the same 1000 queries in order, maximum cache hit happens when there is only one query processor. As we increase the number of query processors, these queries get distributed and processed by different processors, thus cache hit rate generally decreases. This is more evident for our baseline routing schemes, and we find that their throughput saturates with 3$\sim$5 servers. These findings demonstrate the usefulness of smart query routing: *To maintain same cache hit rate, queries must be routed intelligently. Since Embed routing is able to sustain almost same cache hit rate with many query processors (Figure 5(b)), its throughput scales linearly with query processors.*

Storage Tier: We next vary the number of storage servers from 1 to 7, whereas 1 server is used as the router and 4 servers as query processors (Figure 5(c)). When we use 1 storage server, we can still load the entire 60GB *Webgraph* on the main memory of that server, since each of our servers has sufficient RAM. The throughput is the least when there is only one storage server. We observe that 1$\sim$2 storage servers are insufficient to handle the demand created by 4 query processors. However, with 4 storage servers, the throughput saturates, since the bottleneck is transferred to query processors. This is evident from our previous results — the throughput with 4 query processors was about 120 queries per second (Figure 5(a)), which is the same throughput achieved with 4 storage servers in the current experiments.

4.4 Impact of Cache Sizes

In previous experiments, we assign 4GB cache to each processor, which was large enough for our queries; and we never discarded anything from the cache. We next perform experiments when it needs to evict cache entries. In Figure 6, we present average response times with various cache capacities. At the largest, with 4GB cache per processor, no eviction occurs. Therefore, there is no additional performance gain by increasing the cache capacity. On the other extreme, having cache with less than

Landmark	Embed	
	embed per landmark	embed per node
35 sec	36 sec	1 sec

Table 2: Preprocess times

Landmark	Embed	Input graph
2.8 GB	4GB	60.3GB

Table 3: Preprocess storage

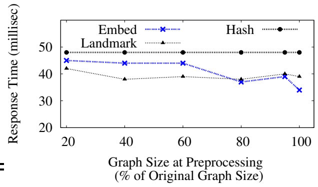
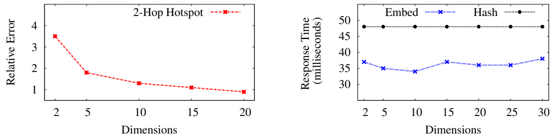


Figure 7: Robustness with graph updates



(a) Relative error

(b) Response time

Figure 8: Impact of embedding dimensionality

64MB per processor results in worse response times than what was obtained with no-cache scheme, represented by the horizontal red line (86ms in Figure 6). When the cache does not have much space, it ends up evicting entries that might have been useful in the future. Hence, there are not enough cache hits to justify its maintenance and lookup costs when cache size $<$ 64MB/ processor.

We also evaluate our routing strategies in terms of minimum cache requirement to achieve a response time of 86ms, the break-even point of deciding whether or not to add a cache. Figure 6(c) shows that smart routing schemes achieve this response time with a much lower cache, as compared to that of the baselines. These results illustrate that *our smart routings utilize the cache well; and for the same amount of cache, they achieve lower response time compared to baseline routings*.

4.5 Preprocessing and Graph Updates

Preprocessing Time and Storage: For landmarks routing, we compute the distance of every node to all landmarks, which can be evaluated by performing a BFS from each landmark. This takes about 35 sec for one landmark in *Webgraph* (Table 2), and can be parallelized per landmark. For embed routing, in addition, we need to embed every node with respect to landmarks, which requires about 1 sec per node in *Webgraph*, and is again parallelizable per node.

The preprocessed landmark routing information consumes about 2.8GB storage space in case of *Webgraph*. On the contrary, with embedding dimensionality 10, the *Webgraph* embedding size is only 4GB. Both these preprocessed information are modest compared to the original *Webgraph* size, which is around 60GB (Table 3).

Graph Updates: In these experiments, we preprocess a reduced subgraph of the original dataset. For example, at 20% of the original dataset (Figure 7), we select only 20% of all nodes uniformly at random, and compute preprocessed information over the subgraph induced by these selected nodes. However, we always run our query over the complete *Webgraph*. We incrementally compute the necessary information for the new nodes, as they are being added, without changing anything on the preprocessed information of the earlier nodes. As an example, in case of embed routing, we only compute the distance of a new node to the landmarks, and thereby find the coordinates of that new node. However, one may note that with the addition of every new node and its adjacent edges, the preprocessed information becomes outdated (e.g., the distance between two earlier nodes might decrease). Since we do not change anything on the preprocessed information, this experiment demonstrates the robustness of our method with respect to graph updates.

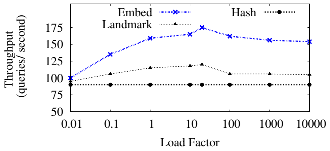


Figure 9: Impact of load factor

Figure 7 depicts that *our smart routing schemes are robust for a small number of graph updates*. With embed routing, preprocessed information over the whole graph results in response time of 34 ms, whereas preprocessed information at 80% of the graph results in response time of 37 ms (i.e., response time increases by only 3 ms). As expected, the response time deteriorates when preprocessing is performed on a smaller amount of graph data, e.g., with only 20% graph data, response time increases to 44 ms, which is comparable to the response time of baseline hash routing (48 ms).

4.6 Sensitivity Analysis

We find that gRouting is more sensitive towards load factor (due to *query stealing*) and embedding dimensionality, compared to other parameters, e.g., number of landmarks and smoothing factor (α). Due to lack of space, we present sensitivity analysis with respect to load factor and embedding dimensionality in Figures 8 and 9. Sensitivity results with other parameters can be found in our extended version [10]. In all figures, we also show our best baseline — hash routing, for comparison.

Embedding Dimensionality: We consider the performance implications of the number of dimensions on embed routing. For these experiments, we create several embeddings, with dimensionality from 2 to 30. While the relative error in distance between node pairs decreases with higher dimensions, it almost saturates after 10 dimensions (Figure 8(a)). On the other hand, we observe that the average response time reduces until dimension 10, and then it slowly increases with more dimensions (Figure 8(b)). This is because with higher dimensions, we reduce the distance prediction error, thereby correctly routing the queries and getting more cache hits. However, a large number of dimensions also increases the routing decision making time at the router. Hence, the least response time is achieved at dimensionality 10.

Load Factor: This parameter impacts both of our smart routing schemes. We find from Equations 3 and 7 that smaller values of load factor diminish the impact of "smart" routing (i.e., landmarks and node co-ordinates), instead queries will be routed to the processor having the minimum workload. On the other hand, higher values of load factor reduces the impact of load balancing (i.e., query stealing) — queries would be routed solely based on landmarks and node co-ordinates. Therefore, in these experiments, we expect that the throughput will initially increase with higher values of load factor, until it reaches a maximum, and then it would start decreasing. Indeed, it can be observed in Figure 9 that with load factor between 10$\sim$20, the best throughput is achieved.

5 Related Work

We studied *smart query routing* for distributed graph querying — a problem for which we are not aware of any prior work. In the following we, however, provide a brief overview of work in neighborhood areas.

Landmarks and Graph Embedding. Landmarks were used in path finding, shortest path estimation, and in estimating network properties [12, 24, 1, 23]. Graph embedding [36] was employed in internet routing, such as predicting internet network distances and estimating minimum round trip time between hosts [4]. To the best of our knowledge, ours is the first study that applies graph embedding and landmarks to design effective routing algorithms for distributed graph querying.

Graph Partitioning, Re-partitioning, Replication. The balanced, minimum-edge-cut graph partitioning divides a graph into k partitions such that each partition contains same number of nodes, and the number of cut-edges is minimized. Even for $k = 2$, the problem is **NP**-hard, and there is no approximation algorithm with a constant approximation ratio unless **P =NP** [18]. Therefore, efforts were made in developing polynomial-time heuristics — METIS, Chaco, SCOTCH, to name a few. More sophisticated graph partitioning schemes were also proposed, e.g., node-cut [6], complementary partitioning [35], and label propagation [33], among many others.

Graph re-partitioning is critical for online queries, since the graph topology and workload change over time [16]. The methods in [35, 18, 11] perform re-partitioning based on past workloads. Incremental partitioning was developed for dynamic and stream graphs [37, 32, 30]. With the proposed embed routing, we bypass these expensive graph partitioning and re-partitioning challenges to the existing cache replacement policy.

Replication was used for graph partitioning, re-partitioning, load balancing, and fault tolerance. In earlier works, [22, 8] proposed one extreme version by replicating the graph sufficiently so that, for every node in the graph, all of its neighbors are present locally. Mondal et. al. designed an overlapping graph re-partitioning scheme [16], which updates its partitions based on the past read/ write patterns. Huang et. al. [7] designed a lightweight re- partitioning and replication scheme considering access locality, fault tolerance, and dynamic updates. While we also replicate the graph data at query processors' cache in an overlapping manner, we *only replicate the active regions* of the graph based on recent workloads. Unlike [16, 7] we do not explicitly run any graph replication strategy at our processors or storage servers. Instead, our *smart routing algorithms* automatically perform replications at processors' cache.

Graph Caching, De-coupling, Multi-Query Optimization. Facebook uses a fast caching layer, Memcached on top of a graph database to scale the performance of graph querying [19]. Graph-structure-aware and workload-adaptive caching techniques were also proposed, e.g., [2, 21]. There are other works on view-based graph query answering [5] and multi-query optimizations [13]. Unlike ours, these approaches require the workload to be known in advance.

Recently, Shalita et. al. [27] employed decoupling for an optimal assignment of HTTP requests over a distributed graph storage. First, they perform a static partition of the graph in storage servers based on co-access patterns. Next, they find past workloads on each partition, and dynamically assign these partitions to query processors such that load balancing can be achieved. While their decoupling principle and dynamic assignment at query processors are similar to ours, they still explicitly perform a sophisticated graph partitioning at storage servers, and update such partitions in an offline manner. In contrast, our smart routing algorithms automatically partition the active regions of the graph in a dynamic manner and store them in the query processors' cache, thereby achieving both load balancing and improved cache hit rates.

6 CONCLUSIONS

We studied h-hop traversal queries – a generalized form of various online graph queries that access a small region of the graph, and require fast response time. To answer such queries with low latency and high throughput, we follow the principle of decoupling query processors from graph storage. Our work emphasized *less* on the requirements for an expensive graph partitioning and re-partitioning technique, instead we developed smart query routing strategies for effectively leveraging the query processors' cache contents, thereby improving the throughput and reducing latency of distributed graph querying. In addition to workload balancing and deployment flexibility, gRouting is able to provide linear scalability in throughput with more number of query processors, works well in the presence of query hotspots, and is also adaptive to workload changes and graph updates.

7 Acknowledgement

The research is supported by MOE Tier-1 RG83/16 and NTU M4081678. Any opinions, findings, and conclusions in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] AKIBA, T., IWATA, Y., AND YOSHIDA, Y. Fast Exact Shortest-path Distance Queries on Large Networks by Pruned Landmark Labeling. In *SIGMOD* (2013).

[2] AKSU, H., CANIM, M., CHANG, Y., KORPEOGLU, I., AND ULUSOY, Ö. Graph Aware Caching Policy for Distributed Graph Stores. In *IC2E* (2015).

[3] BINNIG, C., CROTTY, A., GALAKATOS, A., KRASKA, T., AND ZAMANIAN, E. The End of Slow Networks: It's Time for a Redesign. *PVLDB 9*, 7 (2016), 528–539.

[4] DABEK, F., COX, R., KAASHOEK, F., AND MORRIS, R. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM* (2004).

[5] FAN, W., WANG, X., AND WU, Y. Answering Graph Pattern Queries using Views. In *ICDE* (2014).

[6] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *OSDI* (2012).

[7] HUANG, J., AND ABADI, D. J. Leopard: Lightweight Edge-oriented Partitioning and Replication for Dynamic Graphs. *PVLDB 9*, 7 (2016), 540–551.

[8] HUANG, J., ABADI, D. J., AND REN, K. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB 4*, 11 (2011), 1123–1134.

[9] KARYPIS, G. METIS and ParMETIS. In *Encyclopedia of Parallel Computing*. Springer, 2011.

[10] KHAN, A., SEGOVIA, G., AND KOSSMANN, D. On Smart Query Routing: For Distributed Graph Querying with Decoupled Storage. `https://arxiv.org/abs/1611.03959`, 2016.

[11] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *EuroSys* (2013).

[12] KLEINBERG, J., SLIVKINS, A., AND WEXLER, T. Triangulation and Embedding Using Small Sets of Beacons. *J. ACM 56*, 6 (2009), 32:1–32:37.

[13] LE, W., KEMENTSIETSIDIS, A., DUAN, S., AND LI, F. Scalable Multi-query Optimization for SPARQL. In *ICDE* (2012).

[14] LOESING, S., PILMAN, M., ETTER, T., AND KOSSMANN, D. On the Design and Scalability of Distributed Shared-Data Databases. In *SIGMOD* (2015).

[15] MALEWICZ, G., AUSTERN, M. H., BIK, A. J. C., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A System for Large-scale Graph Processing. In *SIGMOD* (2010).

[16] MONDAL, J., AND DESHPANDE, A. Managing Large Dynamic Graphs Efficiently. In *SIGMOD* (2012).

[17] MONDAL, J., AND DESHPANDE, A. EAGr: Supporting Continuous Ego-centric Aggregate Queries over Large Dynamic Graphs. In *SIGMOD* (2014).

[18] NICOARA, D., KAMALI, S., DAUDJEE, K., AND CHEN, L. Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases. In *EDBT* (2015).

[19] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *NSDI* (2013).

[20] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev. 43*, 4 (2010), 92–105.

[21] PAPAILIOU, N., TSOUMAKOS, D., KARRAS, P., AND KOZIRIS, N. Graph-Aware, Workload-Adaptive SPARQL Query Caching. In *SIGMOD* (2015).

[22] PUJOL, J. M., ERRAMILLI, V., SIGANOS, G., YANG, X., LAOUTARIS, N., CHHABRA, P., AND RODRIGUEZ, P. The Little Engine(s) That Could: Scaling Online Social Networks. In *SIGCOMM* (2010).

[23] QIAO, M., CHENG, H., CHANG, L., AND YU, J. X. Approximate Shortest Distance Computing: A Query-Dependent Local Landmark Scheme. In *ICDE* (2012).

[24] RATTIGAN, M. J., MAIER, M. E., AND JENSEN, D. Using Structure Indices for Efficient Approximation of Network Properties. In *KDD* (2006).

[25] ROY, A., BINDSCHAELDER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out Graph Processing from Secondary Storage. In *SOSP* (2015).

[26] SARWAT, M., ELNIKETY, S., HE, Y., AND MOKBEL, M. F. Horton+: A Distributed System for Processing Declarative Reachability Queries over Partitioned Graphs. *PVLDB 6*, 14 (2013), 1918–1929.

[27] SHALITA, A., KARRER, B., KABILJO, I., SHARMA, A., PRESTA, A., ADCOCK, A., KLLAPI, H., AND STUMM, M. Social Hash: An Assignment Framework for Optimizing Distributed Systems Operations on Social Networks. In *NSDI* (2016).

[28] SHAO, B., WANG, H., AND LI, Y. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD* (2013).

[29] SHUTE, J., VINGRALEK, R., SAMWEL, B., HANDY, B., WHIPKEY, C., ROLLINS, E., OANCEA, M., LITTLEFIELD, K., MENESTRINA, D., ELLNER, S., CIESLEWICZ, J., RAE, I., STANCESCU, T., AND APTE, H. F1: A Distributed SQL Database That Scales. *PVLDB 6*, 11 (2013), 1068–1079.

[30] STANTON, I., AND KLIOT, G. Streaming Graph Partitioning for Large Distributed Graphs. In *KDD* (2012).

[31] TRETYAKOV, K., A.-CERVANTES, A., G.-BANUELOS, L., VILO, J., AND DUMAS, M. Fast Fully Dynamic Landmark-based Estimation of Shortest Path Distances in Very Large Graphs. In *CIKM* (2011).

[32] VAQUERO, L., CUADRADO, F., LOGOTHETIS, D., AND MARTELLA, C. Adaptive Partitioning for Large-scale Dynamic Graphs. In *SOCC* (2013).

[33] WANG, L., XIAO, Y., SHAO, B., AND WANG, H. How to Partition a Billion-Node Graph. In *ICDE* (2014).

[34] WEI, J., XIA, F., SHA, C., XU, C., HE, X., AND ZHOU, A. Workload-Aware Cache for Social Media Data. In *APWeb* (2013).

[35] YANG, S., YAN, X., ZONG, B., AND KHAN, A. Towards Effective Partition Management for Large Graphs. In *SIGMOD* (2012).

[36] ZHAO, X., SALA, A., WILSON, C., ZHENG, H., AND ZHAO, B. Y. Orion: Shortest Path Estimation for Large Social Graphs. In *WOSN* (2010).

[37] ZHENG, A., LABRINIDIS, A., AND CHRYSANTHIS, P. K. Planar: Parallel Lightweight Architecture-Aware Adaptive Graph Re-partitioning. In *ICDE* (2016), pp. 121–132.

Locality-Aware Software Throttling for Sparse Matrix Operation on GPUs

Yanhao Chen[*]
Rutgers University

Ari B. Hayes[*]
Rutgers University

Chi Zhang
University of Pittsburgh

Timothy Salmon
Rutgers University

Eddy Z. Zhang
Rutgers University

Abstract

This paper tackles the cache thrashing problem caused by the non-deterministic scheduling feature of bulk synchronous parallel (BSP) execution in GPUs. In the BSP model, threads can be executed and interleaved in any order before reaching a barrier synchronization point, which requires the entire working set to be in cache for maximum data reuse over time. However, it is not always possible to fit all the data in cache at once. Thus, we propose a locality-aware software throttling framework that throttles the number of active execution tasks, prevents cache thrashing, and enhances data reuse over time. Our locality-aware software throttling framework focuses on an important class of applications that operate on sparse matrices (graphs). These applications come from the domains of linear algebra, graph processing, machine learning and scientific simulation. Evaluated on over 200 real sparse matrices and graphs that suffer from cache thrashing in the Florida sparse matrix collection, our technique achieves an average of 2.01X speedup, a maximum of 6.45X speedup, and a maximum performance loss $\leq$5%.

1 Introduction

Operations on sparse matrix and graph are important for solving linear algebra and optimization problems that arise in data science, machine learning, and physics-based simulation. In this paper we focus on a fundamental sparse matrix operation that relates an input vector x with an output vector y. Let $\mathbf{x}$ be an $n \times 1$ vector, $\mathbf{y}$ be an $m \times 1$ vector, and A be a $m \times n$ matrix, the relation between y and x is defined as $\mathbf{y}$ = A$\mathbf{x}$, where

$$y_i = reduce\_op\{A_{ik} \odot x_k\},\ 1 \leq k \leq n.$$

When the operator *reduce_op* is *sum* and the binary operator $\odot$ is multiplication, the operation is sparse matrix vector multiplication (SpMV). When the operator *reduce_op* is *min* and the binary operator $\odot$ is $+$, the operation is an iterative step in the single source shortest path (SSSP) problem [13]. An example is shown in Figure 1.

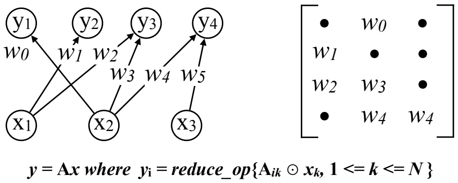


Figure 1: A Fundamental Sparse Matrix Operation

However, poor data reuse is often a problem when running sparse applications on GPUs. Throttling is a useful technique to improve data reuse. Unlike other locality enhancement techniques that focus on spatial data reuse on many-core, for instance, the memory coalescing techniques [33], throttling improves data reuse over time by limiting the number of actively executed tasks.

Throttling prioritizes the execution of the tasks that reuse the data in the cache over those that do not reuse the data in the cache. Figure 2 shows an example of how throttling improves cache data reuse. Assuming the cache capacity is 4, in the original case, the cache cannot hold all the data elements in the execution list which will inevitably cause cache (capacity) misses. Throttling helps by dividing the execution into two phases and scheduling one phase after another. Data elements in each phase can now fit into cache and be fully reused so that no cache (capacity) misses will occur.

Throttling for GPU has been studied extensively in the hardware context. Rogers and others [27] discovered that limiting the number of active wavefronts (warps) enhances cache data reuse over time and alleviates cache thrashing. The DYNCTA framework [19] limits the number of CTAs for memory intensive applications and results in better cache performance. The work by Chen and others [8] augmented cache bypassing with a dy-

[*] First authors Chen and Hayes have made equal contributions to this work and are listed alphabetically.

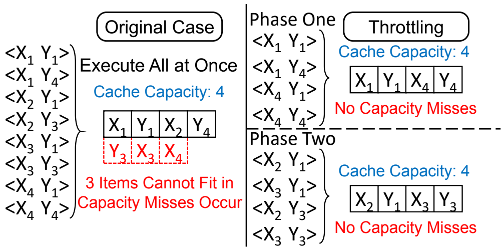


Figure 2: Throttling Example

namic warp-throttling technique to improve both cache performance and energy efficiency. However, all these prior throttling techniques on GPUs have been developed as hardware modifications.

In this paper, we present a software throttling framework that targets irregular applications operating on sparse data. Our software throttling framework will first divide the entire workload into multiple partitions such that the working set of each partition fits into the cache and the data communication between different partitions is minimum (we will refer to each partition as *cache-fit partition* or *cache-fit work group* throughout this paper). Then we schedule the cache-fit partitions and let each of them be processed independently to ensure throttling.

There are three main challenges for realizing software throttling. First, the traditional work partition models focus on minimizing data reuse among different partitions with load-balancing constraints [2, 6, 29]. However, cache-fit work partitioning is not necessarily load-balanced, it should be data-balanced across different partitions. Second, inappropriate scheduling of cache-fit partitions might result in low execution pipeline utilization. For each of the cache-fit partitions that have low data reuse, there may not be enough tasks running concurrently, which will make the execution pipeline units not fully utilized and degrade the computation throughput. Last, reducing the overhead of software throttling is important and yet challenging, especially for finding minimum communication *cache-fit* partitions, which is the most time-consuming step in software throttling.

To tackle these challenges, we propose the three following techniques. To obtain *cache-fit* partitions, we develop an efficient **data-balanced** work partition model. Our partition model can balance data while minimizing the communication cost among different partitions. We also introduce a **split-join** scheduling model to take advantage of the trade-off between throttling and throughput. The *split-join* scheduling model adaptively merges partitions to avoid low execution pipeline utilization and/or use a concurrent queue based implementation for relaxed barrier synchronization. We reduce the partition overhead by a coarse-grained partition model which was built upon a multi-level partition paradigm. Instead of partitioning the original work matrix (graph), our model partitions a coarsened matrix (graph) which can significantly reduce the partition overhead while maintaining similarly good partition quality.

Our throttling technique is a pure software based implementation. It is readily deployable and highly efficient. Evaluated over 228 sparse matrices and graphs from Florida matrix collection [11] - the set of matrices which suffer from cache thrashing (their working set cannot entirely fit into the L2 cache on the Maxwell GPU and Pascal GPU we tested), our software throttling method can achieve an average 2.01X speedup (maximal 6.45X speedup).

As far as we know, this is the first work that systematically investigates **software throttling** techniques for GPUs and is extensively evaluated on real sparse matrices and graphs. The contribution and the outline of our paper is summarized as follows:

- We introduce an analytical model named *data-balanced* work partition for locality-aware software throttling. Efficient heuristics are developed to achieve (near-)minimum communication *cache-fit* work partitions that can be further scheduled to alleviate GPU cache thrashing (Section 2).

- We exploit the trade-off between cache locality and execution pipeline utilization and provide a set of practical *cache-fit* work group scheduling policies based on *adaptive merging* and *concurrent dequeuing*. We discuss the advantages/disadvantages, the applicability, and the effectiveness of each scheduling policy in different settings. (Section 3).

- Our method requires *no hardware modification*. It is low overhead and readily deployable. We introduce efficient overhead control mechanisms for graph(matrix)-based work partition. (Section 4).

- We conduct a *comprehensive data analysis* for over 200 large real sparse matrices(graphs). Our framework in particular works well for the set of sparse matrices that have large working sets and suffer from high GPU cache contention (Section 5).

2 Data-Balanced Work Partition

Our software throttling framework first divides the entire workload into cache-fit partitions. A cache-fit partition's working set fits into the cache such that it will not cause any cache capacity miss. This section presents the concept and methodology of data-balanced work partition.

2.1 Graph Representation

In this paper, we focus on a fundamental operation in sparse linear algebra and optimization applications. It is defined as follows. Assume we have an $m \times n$ matrix A, an $n \times 1$ vector x, and an $m \times 1$ vector y such that:

$$y_i = reduce\_op\{A_{ik} \bigodot x_k\}, 1 \leq k \leq n \qquad (1)$$

The operator $\odot$ is a binary operator, and the operator *reduce_op* is a reduction operator. When $\odot$ is product $\times$ and *reduce_op* is *sum*, the operation is a sparse matrix vector multiplication (SpMV). When $\odot$ is plus $+$ and *reduce_op* is *min*, the operation is a *min/product* step in the single source shortest path (SSSP) problem.

We represent a computation unit as a 2-tuple (x_j, y_i) which represents (1) one binary $\odot$ operation between x_j and A_{ij} , and (2) one step in the reduction operation *reduce_op* for obtaining y_i. We only focus on vector x and y, since the matrix entries will be used only once in Equation (1).

We represent the entire workload as a *2-tuple* list. Using a graph representation, each data element in the *2-tuple* is modeled as a vertex and each tuple is modeled as an edge that connects the corresponding two vertices. Performing a work partition is essentially performing an edge partition on the graph, as illustrated in Figure 3.

2.2 Data-Balanced v.s. Load-Balanced

We formally define the **data-balanced** *work partition* model. The input is a list of *2-tuple* modeled as a work graph and the output is a set of minimum-interaction work partitions such that the number of unique vertices, which represent data elements, in every work partition is less than or equal to the cache capacity.

In contrast to prior load-balanced work partition, we perform data-balanced work partition. We denote this problem as a *Vertex-balanced Edge-Partition (V-EP)* model and we give the definition below:

Definition 2.1.
Vertex-balanced Edge-Partition (V-EP) Problem *Given a graph $G = (V,E)$ with the set of vertices V and the set of edges E, and vertex capacity constraint T. Let $x = \{e_1, e_2, ...e_k\}$ denote a partition of the edges of G into k disjoint subsets, and let $V(e_i)$ denote the set of unique vertices in e_i. $\forall n \in V$, let $P(n)$ denote the number of subsets that n's incident edges fall into. We optimize the total vertex replication cost:*

$$\begin{aligned} \underset{x}{minimize} \quad & R(x) = \sum_{n \in V} (P(n) - 1) \\ subject\ to \quad & \forall i \in [1..k], |V(e_i)| \leq T \end{aligned} \qquad (2)$$

In prior work, the *Edge-balanced Edge-Partition* (E-EP) problem has been well studied particularly in the distributed graph processing setting [6, 14] and also for balancing workloads in GPU [25, 26]. However, the V-EP problem is not. Both the V-EP and E-EP problems minimize vertex replication cost, while the E-EP model aims to balance the load among processors in space, and the V-EP model aims to alleviate cache thrashing and maximize data reuse over time.

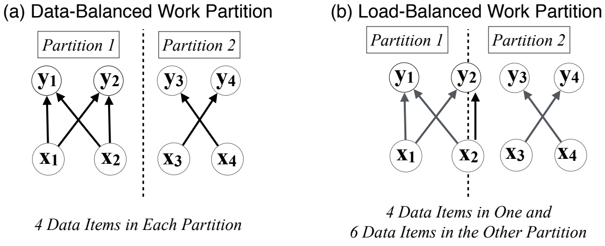

(a) Data-Balanced Work Partition
(b) Load-Balanced Work Partition
Partition 1
Partition 2
y1
y2
y3
y4
x1
x2
x3
x4
Partition 1
Partition 2
y1
y2
y3
y4
x1
x2
x3
x4
4 Data Items in Each Partition
4 Data Items in One and
6 Data Items in the Other Partition

Figure 3: Data-Balanced v.s. Load-Balanced

We use an example in Figure 3 to illustrate the difference between the V-EP work partition and the E-EP work partition. Assuming the cache capacity is 4, Figure 3 (a) shows a 2-way V-EP work partition: one partition has 4 edges and the other has 2, the unique vertices of both (4 vertices) fit into cache. Figure 3 (b) shows another 2-way E-EP partition: Each partition has 3 edges, however *partition 2* has 6 unique vertices and do not fit into cache. Thus the E-EP model might exacerbate rather than alleviate the cache thrashing problem.

2.3 Partition Framework

We propose a data-balanced work partition framework that ensure the working set of each partition is of the same size and in the meantime the data reuse across different partitions is reduced as much as possible.

Our partition framework is a recursive *bisection* framework. Bisection is a *2-way* balanced edge partition that ensures minimum vertex replica between two equal-size edge partitions. The optimal *bisection* is a well studied problem [25]. We take the advantage of the *bisection* method and perform hierarchical partitioning.

During the recursive partition process, the framework bisects a sub-graph that has more unique vertices than specified by the capacity constraint. It keeps bisecting until no such sub-graph exists.

We use a tree data structure to keep track of the obtained sub-graphs. Starting from the root node that represents the entire work graph, the framework bisects the corresponding graph and generates two children nodes: each of the child nodes corresponds to a sub-graph that contains half of the edges from the parent node. If either or both children nodes violate the capacity constraint, either or both will be added to the list of sub-graphs that need to be further bisected. The process repeats until all leaf nodes become cache-fit work partitions.

The detailed algorithm is listed below in Algorithm 1. The *data-balanced work partition* (DBWP) procedure takes the work graph G and the cache capacity constraint T as input, and generates a set of cache-fit partitions P as output. The *bisect* function in Algorithm 1 we adopted is based on the best existing balanced edge partition algorithm named SPAC [26, 24] by Li and others. We will discuss the implementation details and the overhead control mechanisms of the *bisect* function in Section 4.

Algorithm 1 Data-Balanced Work-Partition (DBWP)

```
Input: work graph G, cache capacity T
Output: cache-fit partition set P
procedure DBWP(G, T, P)
    if $|G.data\_elements| > T$ then
        (lchild, rchild) = bisect(G)
        DBWP(lchild, T, P)
        DBWP(rchild, T, P)
    else
        add G to P
    end if
end procedure
```

We use an example to illustrate the *DBWP* procedure in Figure 4. In this example, the graph has 8 edges and 8 vertices, and the cache capacity constraint is 4. Performing *bisect* for the sub-graph represented by the tree root node, we obtain two sub-graphs each of which has 4 edges. The vertex replica cost is optimum: 2, since two nodes y_2 and x_2 appear in both partitions.

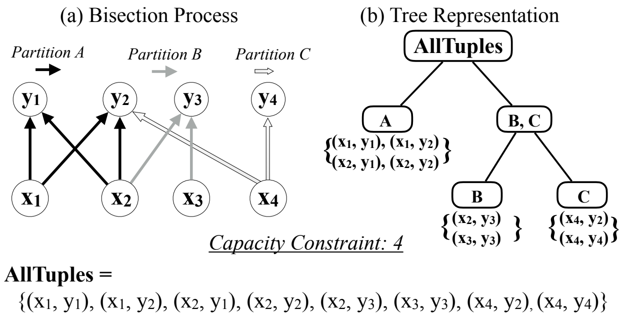


Figure 4: Hierarchical Bisection Example

The first sub-graph A in Figure 4 (a) has 4 unique vertices and does not violate the capacity constraint, we do not perform further bisection on sub-graph A. The other sub-graph, however, has 6 unique vertices and does not fit into the cache. Therefore we perform the second bisection and obtain partitions B and C where the vertex replica cost is optimum (0 in this case). At this point, there is no sub-graph that does not fit into the cache, therefore we terminate the bisection process. The tree representation is shown in Figure 4 (b).

3 Cache-Fit Partitions Scheduling

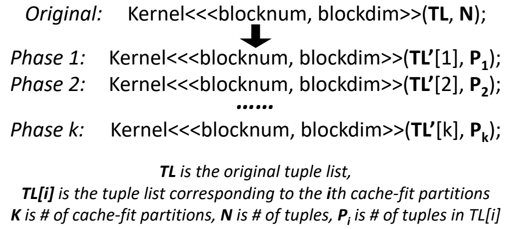
DBWP model outputs a set of *cache-fit* partitions. All these partitions need to be processed independently to minimize cache-thrashing interference. However, naive scheduling of these partitions might result in low execution pipeline utilization. In this section, we introduce four different *Cache-Fit Partitions* Scheduling methods: *Cache-Fit* Scheduling (**CF**), *Cache-Fit Queue* Scheduling (**CF-Q**), *Split-Join* Scheduling (**SJ**) and *Split-Join Queue* Scheduling (**SJ-Q**).

CF works well when all cache-fit partitions have high data reuse. **SJ** is good for cases when the sparsity structure is already known, for instance, pruned deep leaning neural networks. Both **CF-Q** and **SJ-Q** can loosely enforce throttling and provide a better performance.

3.1 Cache-Fit Scheduling

A straightforward way to isolate the computation of different cache-fit partitions is to assign each partition a single *kernel* function and execute these kernels one by one. A *kernel* is a function that is executed on GPU. All threads within a GPU kernel will need to finish before the entire kernel complete – there is a strict barrier between different GPU kernels. Moreover, between two consecutive kernels, the data in the cache will be invalidated.

Here, we propose **CF** which separates the original kernel functions into multiple kernels, while the number of which is determined by the number of cache-fit partitions given by **DBWP** model. The code of the kernel function for each cache-fit partition is the same. The only difference is the input to each kernel. **CF** ensures that the data in the cache is fully reused before it was evicted from the cache within each cache-fit partition.



```
Original:   Kernel<<<blocknum, blockdim>>(TL, N);
                          ⬇
Phase 1:    Kernel<<<blocknum, blockdim>>(TL'[1], P1);
Phase 2:    Kernel<<<blocknum, blockdim>>(TL'[2], P2);
                        ……
Phase k:    Kernel<<<blocknum, blockdim>>(TL'[k], Pk);
```

TL is the original tuple list,
TL[i] is the tuple list corresponding to the ith cache-fit partitions
K is # of cache-fit partitions, N is # of tuples, P_i is # of tuples in TL[i]

Figure 5: Kernel Splitting for Cache-Fit Scheduling

We show the idea of **CF** in Figure 5. The input *2-tuple* task list TL is split into k *2-tuple* lists TL', which corresponds to each of the cache-fit partitions. Each new tuple list will be processed by a single kernel.

3.2 Cache-Fit Queue Scheduling

CF makes sure that each cache-fit partition will be processed by one kernel. Although **CF** can provide good throttling performance for a lot of matrices, this scheduling method may cause low execution pipeline utilization for the type of matrices whose data reuse is low. For example, for a given cache size T and average data reuse ratio r for a cache-fit partition, the total work is $T * r/2$ using our work graph model. The variable T is fixed for a given architecture. If r is low, the number of concurrent tasks in one cache-fit partition p is low and may not keep the execution pipeline busy.

To avoid this problem, we propose **CF-Q**, which processes the whole tuple list in a single kernel instead of one invocation per cache-fit partition. However, using a single kernel means that elements in one cache-fit partition have no guarantee to be executed without any interference. To enable throttling, we set up a FIFO queue before launching the kernel. Each queue entry corresponds to a chunk of tuples so that adjacent chunks are from the same cache-fit partition. A warp automatically fetches a chunk from the queue and process the tuples from that chunk. We show an example of how **CF-Q** works in Figure 6.

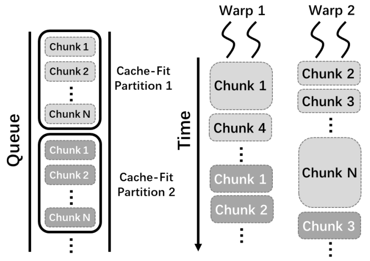


Figure 6: Queue Based Scheduling Example

Unlike **CF** which has explicit barriers to strictly enforce the independent execution of different cache-fit partitions, **CF-Q** uses no barrier. It is possible that the last chunk in one partition and the first chunk in the next partition are fetched within a very short time period. In Figure 6, chunk 1 and chunk 2 from partition 2 will be running concurrently with chunk N from partition 1. However, **CF-Q** can still provide relaxed barriers between different partitions since chunks from the same cache-fit partition in the queue will always be retrieved in consecutive time periods so that no following partitions can be executed before previous one starts. The pseudo code of **CF-Q** is provided in Algorithm 2.

Algorithm 2 Cache-Fit Queue Scheduling

```
Input: cache-fit partition set $P$
procedure CF-Q($P$)
    for each partition $p$ in $P$ do
        insert $p$ into queue $Q$
    end for
    Kernel($Q$)
end procedure
procedure KERNEL($Q$)
    while Q is not empty do
        $I \leftarrow$ next queue item (chunk) from $Q$
        process $I$[laneID]
    end while
end procedure
```

3.3 Split-Join Scheduling

Split-Join (**SJ**) is another method that exploits the tradeoff between locality and execution pipeline utilization. **SJ** dynamically merges the *cache-fit* partitions that has low data reuse or combines low data reuse partitions with a high data reuse partition that is less likely to be interfered. **SJ** first constructs the tree structure that represents the hierarchical cache-fit partitions which we discussed in Section 2.3, we will refer to this as *SJ-tree*. **SJ** merges sibling nodes in the *SJ-tree* conditionally in a bottom-up manner. We only consider recombining sibling nodes as the sibling nodes have better data sharing than non-sibling nodes and the merging is logarithmic time.

SJ is performed with a fast online profiling process. We define a *profiling pass* as a profiling of all the nodes at one level of the *SJ-tree* which comprises one traversal of the entire work graph. So, the entire profiling process will take d profiling passes, where d is the depth of the *SJ-tree*. It takes at least $log(k)$ and up to k profiling passes for any given *SJ-tree*, where k is the number of leaf nodes in the tree. The lower bound $log(k)$ is reached when the binary tree is balanced and has $log(k)$ levels. Moreover, in the worst case scenario, when the tree is not balanced and at every level there is at most one leaf node, k profiling passes are needed. We run every work partition that corresponds to a leaf node in the *SJ-tree* in *stand-alone* mode and record the running time.

We use the first d iterations of the linear algebra and optimization applications to collect information for profiling. Since those applications we tested take between 50 and 22,000 iterations to converge, the overhead of profiling can be amortized. For example, *G3_circuit* need 5 iterations for profiling, and the total profiling time for Conjugate Gradient (CG) solver takes 0.017 s. The running time for CG with 22824 iterations is 94.331 s which gives us 0.018% profiling overhead. In practice, among all the matrices we used in the experiment, we found that the SJ-tree had at most 8 levels and thus required at most 8 passes for profiling.

The tree node merging problem can be defined as a tree-based weighted set cover problem. Merging two sibling nodes is as if choosing their parent node. The problem becomes how to find a subset of tree nodes $\boldsymbol{P}$ that will cover all possible cache-fit partitions (leaf nodes) while minimizing the overall running time:

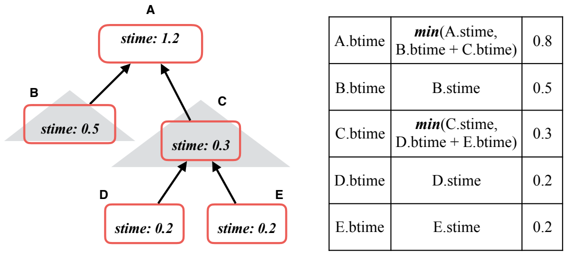
$$\begin{aligned} \text{minimize} \quad & \sum_{x \in P} c(x) \\ \text{subject to} \quad & \bigcup_{x \in P} S(x) = L \end{aligned} \tag{3}$$

where L is the set of all leaf nodes, P is the subset of the tree nodes (both leaf and non-leaf nodes) we want to find, $c(x)$ is a cost function that denotes the standalone running time of node x and S(x) is a set function that returns all leaf nodes of the subtree under node x.

Algorithm 3 Tree Recombination

```
Input: SJ-tree root
Output: optimal running time of SJ-tree root
procedure TREERECOMB(root)
    return BESTCONFIG(root)
end procedure
procedure BESTCONFIG(r)
    left_t = BESTCONFIG(r.leftChild)
    right_t = BESTCONFIG(r.rightChild)
    this_node_t = r.stime
    r.btime = min(left_t + right_t, this_node_t)
    return r.btime
end procedure
```

We develop a linear time algorithm that is capable of finding the optimum solution for the *tree-based set cover* problem. The algorithm processes the tree in post-topological order. Every node is associated with an attribute *btime* and an attribute *stime*. A sub tree's optimum time *btime* (annotated as an attribute of its root node) is the minimum of the two items: its root node's standalone running time *stime* and the summation of its two children subtree's *btime*. For a leaf node, its *btime* is the same as its standalone running time *stime*. The pseudo code is provided in Algorithm 3 together with an example in Figure 7. This process identifies the best set cover of the SJ-tree and determines how to recombine cache-fit partitions into every GPU kernel. **SJ** can achieve high execution pipeline utilization without sacrificing cache benefits (as data reuse is low for these low pipeline utilization cases).



A.btime	*min*(A.stime, B.btime + C.btime)	0.8
B.btime	B.stime	0.5
C.btime	*min*(C.stime, D.btime + E.btime)	0.3
D.btime	D.stime	0.2
E.btime	E.stime	0.2

{B, C} is chosen as the best configuration.

Figure 7: Tree Node Recombination Example

3.4 Split-Join Queue Scheduling

SJ dynamically merges cache-fit partitions that has low data reuse to ensure high execution pipeline utilization and good throttling performance. However, although **SJ** can provide Strict Barriers between different (merged) partitions, **SJ** cannot guarantee the execution order of those cache-fit partitions inside the merged partitions.

We propose **SJ-Q** which uses the idea of **CF-Q** that places cache-fit partitions in one merged work group (kernel) into a queue and each kernel will be using one independent queue. **SJ-Q** can provide both strict barriers between different merged partitions and also relaxed barriers between cache-fit partitions from the same merged partition. In the mean time, it inherits the advantage of **SJ** that avoids low execution pipeline utilization.

Sched.	Pipeline Util.	Prof.	Barrier	Queue	Code Change
CF	Low	No	Strict (S)	No	No
CF-Q	High	No	Relaxed (R)	Yes	Yes
SJ	High	Yes	Strict	No	No
SJ-Q	High	Yes	S/R	Yes	Yes

Table 1: Comparison of Four Scheduling Methods: Sched. refers to Scheduling Method, Prof. refers to if profiling is needed, and Util. refers to utilization.

Summary CF enforces strict barriers between different cache-fit partitions to ensure throttling. However, low execution pipeline utilization may happen which can degrade the computation performance; **CF-Q** uses a queue based method that can fully utilize the execution pipeline and loosely enforce barriers between consecutive cache-fit partitions; **SJ** merges those low data reuse partitions into one based on a tree set cover algorithm and online profiling; **SJ-Q** enforces strict barriers between different merged partitions and relaxed barriers between different cache-fit partitions within one merged partition.

We summarize the features of four scheduling methods in Table 1. All methods ensure good throttling performance while different methods impose different levels of barrier synchronization and code change overhead.

4 Implementation

We perform the adaptive overhead control mechanisms and data reorganization to make our software throttling method more efficient. We reduce the overhead of bisection in the **DBWP** Model, which is the most time-consuming part. In particular, we focus on two bisection algorithms: 1) Coarsened Bisection built upon the SPAC model by Li and others [25], and 2) K-D tiling built upon the k-d tree geometric space partitioning method [5].

We also use CPU-GPU pipelining to make the scheduling overhead transparent [33]: the CPU determines the best schedule while GPU is doing the actual computation. We improve the kernel performance by transforming data layout after we get cache-fit partitions such that the data access within the same kernel is coalesced as much as possible.

4.1 Adaptive Overhead Control

Coarsened Bisection *Coarsened Bisection* is based on SPAC [25] an effective sequential edge partition model. SPAC relies on a multi-level partition paradigm, in which, a graph is coarsened, partitioned, and refined/uncoarsened level by level. In *Coarsened Bisection*, we reduce the overhead of SPAC by eliminating the last few levels of refinement steps. We discovered that the last few coarsened levels (five to seven levels) of refinement stages in the multilevel partitioning scheme, if omitted, do not lead to much performance difference for our software throttling methods. While at the same time, a large amount of partition over can be saved.

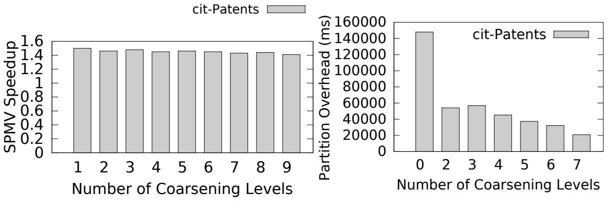


Figure 8: Trade-off between Eliminated Refinement Levels and Scheduling Overhead/Benefits

We show the trade-off between the number of levels where the refinement step is eliminated, and the SPAC partition overhead, and the *SpMV* speedup when applying the SPAC for scheduling, for the sparse matrix cit-patents [11] in Figure 8. It can be seen from the figure that by eliminating the last five to seven levels of refinement, the overhead is reduced by up to 7.5x, while the *SpMV* speedup only changed from 1.5x to 1.4x.

The detailed algorithm is showed in Algorithm 4. Notice that the input graph is already an coarsened graph, since we can perform first level coarsening while reading data from file. We also parallelize the merging phase in the coarsening phase to further reduce overhead. The merging phase is mainly for reconstructing the coarsened graph in each level and is amenable to parallelization.

Algorithm 4 Coarsened Bisection

```
Input: Coarsened Graph $G$
Output: Partition $P$
procedure COARSENBISECT($G$)
    // we call a set of edges - an entity
    build entity based adjacent list $L$ of $G$
    for $level \in \{1, \ldots, maxLevel\}$ do
        sort $L$ by entity degree
        for each entity $e$ do
            merge $e$ with its heaviest avaliable neighbor $ne$
        end for
        build coarsened $L$ by results from above step
    end for
    build coarsened graph $G'$ from $L$
    $P \leftarrow$graphPartition($G'$)
end procedure
```

K-D Tiling Another bisection method we adopted is a tiling based method: *K-D Tiling*. Since any graph can be converted into a sparse matrix representation, we treat the partition as a partition in a geometric two-dimensional space. This method is similar to the k-d tree structure [5] used for partitioning a k-dimensional space. Every non-leaf node in a k-d tree represents a division point along a single spatial dimension, recursively splitting the space's points into two groups.

This partitioning method has even lower overhead than *Coarsened Bisection*. Each split can be performed in O(n) average time via the *quickselect* algorithm [16], and the number of rounds of splitting is logarithmic. However, unlike *Coarsened Bisection*, the tiling approach does not consider connectivity of the graph, and so it generates inferior results. This trade-off makes *Coarsened Bisection* preferable in applications where its overhead can be hidden via amortization, for instance, in optimization problems, and the K-D tiling method is better for overhead-sensitive applications.

4.2 Data Reorganization

After we perform *Data-Balanced Work-Partition* on the work graph, we reorganize the data in memory according to cache-fit partitions for efficient memory coalescing. We prioritize the partition that has the smallest amount of unique data – indicating a high data reuse if the amount of work in each partition is the same. We iterate over each partition's tuple list, and place all their non-boundary vertices (vertices that only appear in one kernel) consecutively in memory using *data packing* [12]. After non-boundary vertices for each partition have been processed, we process boundary vertices. The data reordering algorithm is briefly described in Algorithm 5.

5 Evaluation

We perform experiments on two platforms: an NVIDIA *GTX 745* GPU with Intel Core i7-4790 CPU and an NVIDIA *TITAN X* GPU with Intel Xeon CPU E5-2620. The GPU configurations are detailed in Table 2. We evaluate our techniques using important real-world workloads including sparse linear algebra, neural networks, and graph analytics.

Table 2: Experimental Environment

GPU Model	Titan X	GTX 745
Architecture	Pascal	Maxwell
Core #	5376	576
L2 Cache	3MB	2MB
CUDA version	CUDA 8.0	CUDA 8.0

Algorithm 5 Data Remapping

```
Input: Original Partition Set $P$, Boundary Vertex Set $B$
Output: Reordered Data $D$
procedure DATAREMAPPING($P, D$)
    for each vertex $v$ in partition $p$ of $P$ do
        if $v \notin B$ then
            $unique[p]$++
        end if
    end for
    P' = rank(P, $unique$[P]); // Rank $P$ by $unique$[]
    // Assign non-boundary nodes
    for each vertex $v$ in partition $p$ of $P'$ do
        if !$boundary[v]$ and $v$ is not in $D$ then
            append $v$ to $D$
        end if
    end for
    // Assign boundary nodes
    for each vertex $v$ in partition $p$ of $P'$ do
        if $boundary[v]$ and $v$ is not in $D$ then
            append $v$ to $D$
        end if
    end for
end procedure
```

Sparse Linear Algebra Workloads We use sparse matrix vector multiplication (SpMV) and the conjugate gradient solver (CG). We present performance and sensitivity analysis, as well as the effectiveness of overhead control.

Neural Networks We use a pruned form of AlexNet [15]. The pruned neural network is essentially sparse matrix operation.

Graph Processing Workloads We use two graph processing benchmarks: the Bellman-Ford (BMF) and PageRank (PR) programs [21]. Bellman-Ford takes a weighted graph as input and iteratively calculates every node's distance – an important, basic operation used in path and network analysis applications. PageRank takes a weighted graph as input and calculates the importance of every node based on its incoming links.

Computational Fluid Dynamics Workloads We use the CFD benchmark from the Rodinia benchmark suite [7]. The CFD solver is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow. The CFD benchmark is already highly optimized in terms of data layout [9]. We use three mesh input sets from Rodinia [7].

5.1 Sparse Linear Algebra

SpMV We treat the sparse matrix as a bipartite graph, as described in Section 2.1, and then apply our techniques . We use the SpMV kernel function from the cusp library[10] and the matrix format is COO.

Of the 2757 matrices in the University of Florida collection [11], we extract those where the working set cannot fit entirely into the L2 (last-level) cache, which leaves us with 228 matrices on GTX 745, and 192 on Titan X. Though we optimize for the L2 cache, our techniques can be generalized to other caches.

The performance summary for *SpMV* across these matrices is shown in Figure 9. We also include Org+R, which applies the data reorganization scheme described in Section 4.2 to the original program to optimize memory coalescing. Memory coalescing is a technique for enhancing spatial locality [33], which is orthogonal to our technique proposed in this paper. As our work also performs memory coalescing after obtaining and scheduling cache-fit partitions, we show the performance of *memory coalescing only* (Org+R) versus our technique + *memory coalescing* for fair comparison and for demonstrating the significant performance improvement from our technique.

Among the other methods shown, first is CF, the Cache-Fit method described in Section 3.1. This splits the kernel to run each of the cache-fit partitions in standalone mode. Second is SJ, the Split-Join method described in Section 3.3 and uses tree-based set cover algorithm to merge cache-fit partitions. Third is CF-Q from Section 3.2, which applies the concurrent queue for loosely enforcing cache-fit partition ordering within a single kernel invocation. Finally we show SJ-Q from Section 3.4, which applies the concurrent queue to merged partitions in SJ. We use our Coarsened Bisection partitioner for all four of these methods. The baseline is the original program performance.

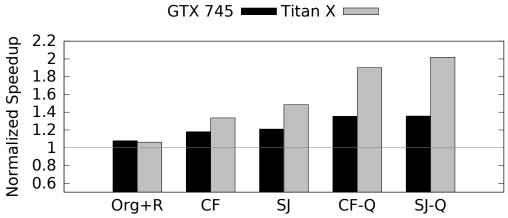
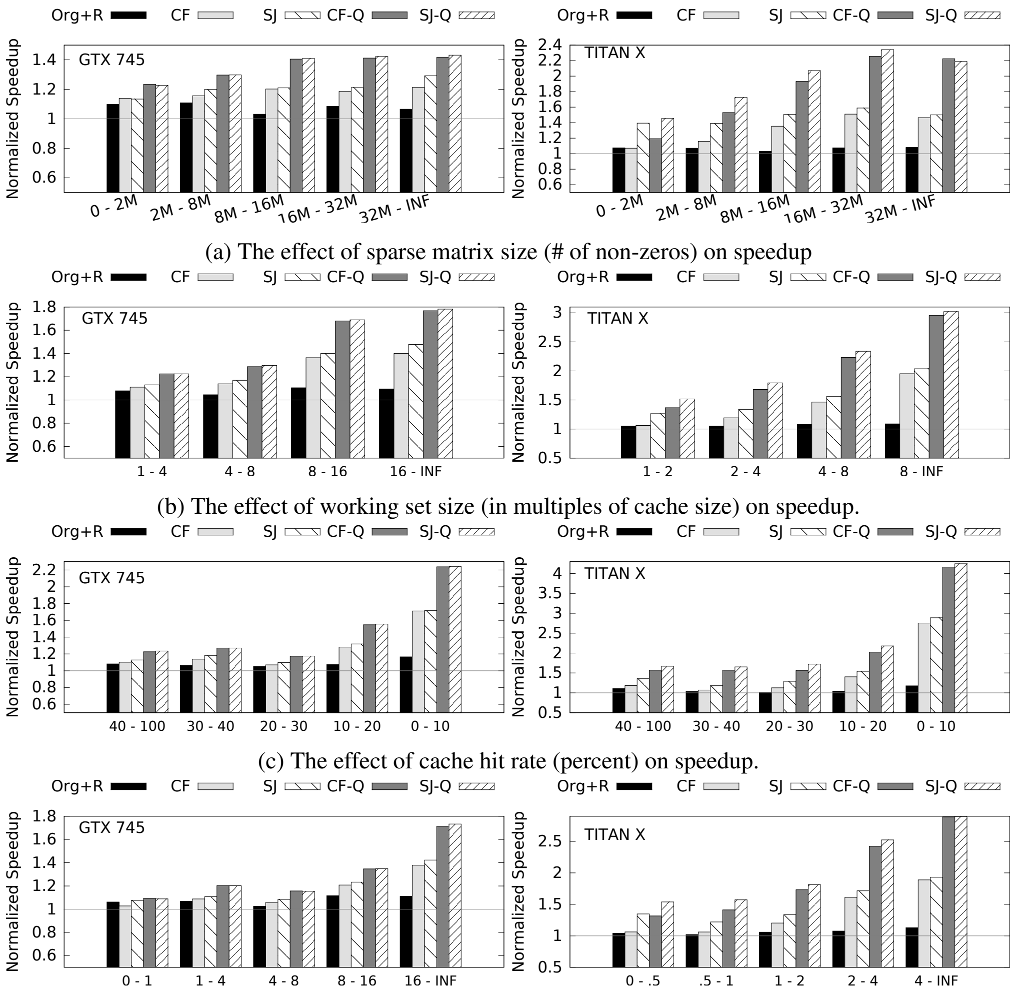


Figure 9: Average Speedup for SpMV

All four methods provide significant speedup compared to the original case and the Org+R case. But each method has trade-offs. The SJ and SJ-Q methods both require runtime profiling, whereas CF and CF-Q do not need runtime profiling. CF and SJ are easier to incorporate to a program as the kernel code does not change (only the input to each kernel invocation changes), whereas CF-Q and SJ-Q require code modification in order to implement the queue.

We find that our techniques provide significantly more improvement in the high contention environments of larger matrices with lower hit rates. We demonstrate the effectiveness of these methods with respect to matrix size, working set size, cache hit rate, and original running time.

Matrix Size In Figure 10 (a), each group of bars shows average speedup for sparse matrices with the specified amounts of non-zeros. Every method except Org+R is



(a) The effect of sparse matrix size (# of non-zeros) on speedup

(b) The effect of working set size (in multiples of cache size) on speedup.

(c) The effect of cache hit rate (percent) on speedup.

(d) The effect of original runtime (milliseconds) on speedup.

Figure 10: SpMV Speedup on GTX 745 and Titan X

much more effective on larger matrices than on smaller ones, but we do see speedup in every group.

Working Set Our techniques become more effective as the working set grows, alleviating the increased cache contention. In Figure 10 (b), each group of bars shows average speedup for matrices with a working set of specified size. The unit used for the x-axis is the number of times the working set can completely fill the cache.

We see speedup improve as the working set grows, just as it tends to do when the matrix size grows. But the effects are more pronounced, with higher speedup. This shows working set size is more useful than matrix size for determining whether we should use locality-aware software throttling optimization.

Cache Hit Rate Our techniques are designed to improve matrices that suffer from low hit rates due to cache thrashing. As such, a lower original hit rate allows us to achieve higher speedup. In Figure 10 (c), each group of bars shows average speedup for matrices with a specified range of cache hit rates for the original case.

For matrices with lowest hit rates, the speedup for the queue-based approaches is particularly extreme. This shows that the queue-based approach is especially effective in environments that have high cache contention. The implicit communication between thread warps competing for queue reservations allows warps to achieve higher temporal locality with each other.

Run Time In Figure 10 (d), each group of bars shows the average speedup for matrices with the specified original runtime, measured in milliseconds. Since the Titan X device is much faster than the GTX 745, we use smaller thresholds for it. In general we can expect that the runtime correlates highly with the number of non-zeros, and so this figure shows a similar curve to the others.

CG We show the performance of conjugate gradient (CG) using SJ-Q. The major computation component is sparse matrix vector multiplication (SpMV). It calls *SpMV* iteratively until convergence. Therefore the overhead is amortized across different iterations. We show the overall performance in Figure 11 for a representative set of inputs. We find the performance improvement of CG with overhead is similar to that of *SpMV* without overhead. The overhead of Coarsened Bisection and SJ-Q profiling is well amortized.

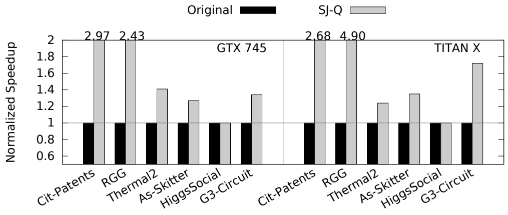


Figure 11: CG Speedup on GTX 745 and Titan X

We also show the L2 cache hit rates for CG in Figure 12. The changes to cache hit rates correlate with the performance improvement. The matrix rgg_n_2_23_s0 (RGG) has a much smaller cache hit rate on Titan X (0.44%) than on GTX 745 (36.75%), despite its larger cache size. There is more cache contention on Titan X since it uses more cores. We are able to improve the hit rate to 62.92% without changing the thread number or the implementation of the kernel code. Only the set of non-zero elements processed by each kernel is changed.

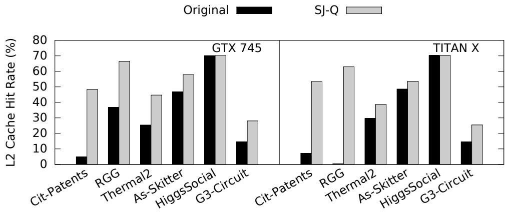


Figure 12: CG Cache Hit Rate on GTX 745 and Titan X

5.2 Neural Networks

We explore the effectiveness of our techniques on the AlexNet neural network, achieving an overall speedup of up to 54% on the Titan X device, which is suited for deep learning. Each fully connected layer of the neural network AlexNet operates as a matrix-vector operation; the matrix is a weight matrix. The work by Han and others [15] prunes the AlexNet network to remove elements of low weight and result in sparse matrices.

Since AlexNet is designed for smaller, embedded devices, we run multiple instances in parallel, allowing the neural network to analyze 150 different images at once for our Titan X GPU. This provides a reasonable amount of computation and data for our more powerful hardware.

In Figure 13, we show the speedup achieved by our technique on each of the three pruned *fc* layers of AlexNet. We include the alternate baseline of the original case plus data reorganization, as well as the Cache-Fit and Split-Join strategies both with and without the queue-based implementation.

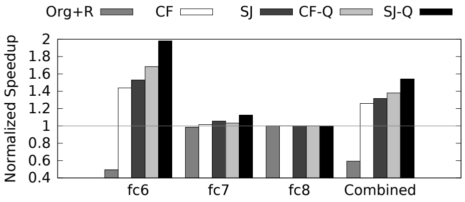


Figure 13: Speedup for AlexNet layers on Titan X

When only applying data reorganization, we see no improvement or even some slowdown. But when we apply any of our partitioning techniques we see speedup up to 98%, and no degradation on the smaller layers. The reason for less-speedup in the smaller layers (fc 7 and fc 8) is that their vector size is smaller and can fit into last level cache entirely in our Titan X GPU. We believe the performance improvement will be more pronounced for Alexnet if we test with embedded devices.

5.3 Graph Applications

We show the performance of the Bellman-Ford and PageRank programs on a set of graphs from the University of Florida Sparse Matrix Collection [11], Stanford Large Network Dataset Collection [23], and DIMACS implementation challenge [1]. Information for each graph is listed in Table 3.

We demonstrate the efficiency of the K-D tiling (SJ-kdtiling) approach, since both BMF and PR take fewer iterations to converge compared with sparse linear system solvers. Thus we need a fast and approximate partitioner so that the overhead can be amortized. We use SJ rather than SJ-Q, since it still provides good speedup while avoiding the overhead of the queue.

We summarize the performance with overhead in Table 3. We see that our approach improves performance for both BMF and PR. *RoadCal* benefits least, due to small size, but sees improvement in some cases.

Table 3: BMF and PR Performance Summary

Graph	GTX 745		TITAN X	
	BMF	PR	BMF	PR
Pokec [23]	1.62	2.98	1.88	3.17
WebGoogle [23]	2	3.29	1.8	3.37
Wikipedia-051105 [11]	1.24	1.99	1.43	1.97
WikiTalk [11]	1.74	2.57	2.09	2.75
IMDB [11]	2.16	3.22	1.59	2.62
RoadCentral [11]	1.19	1.6	1.69	2.18
RoadCal [1]	1	1	1	1.22

We observe that the speedup for PR is greater than for BMF. There are more memory accesses in the PR algorithm than in the BMF algorithm, and so it benefits more from our locality-aware software throttling.

We show cache hit rates for each program in Figure 14 and Figure 15. We show speedup with and without overhead for PR on Titan X in Table 4. PR has fewer iterations than CG so cannot improve performance with Coarsened Bisection if overhead is considered. However, the KD-Tiling method is fast enough that for SJ-kdtiling's performance to remain high with overhead.

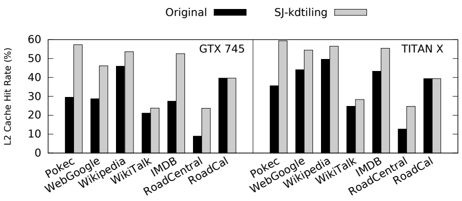


Figure 14: Cache Hit Rates for Bellman-Ford

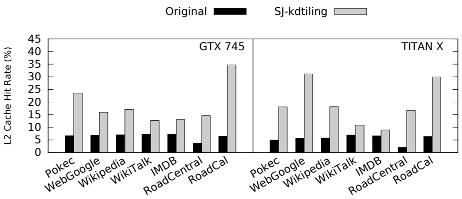


Figure 15: Cache Hit Rates for PageRank

Table 4: PageRank Speedup with and without Overhead

Graph	SJ-kdtiling w/ Overhead	SJ-kdtiling w/o Overhead
Pokec	3.17	3.34
WebGoogle	3.37	3.43
Wikipedia	1.97	1.98
WikiTalk	2.75	2.81
IMDB	2.62	2.27
RoadCentral	2.18	2.48
RoadCal	1.22	1.21

5.4 Computational Fluid Dynamics

The graph structure for CFD is a mesh in which every node has up to four neighbors. Since these meshes are small, We use SJ instead of SJ-Q for throttling. In Figure 16 we show the performance on GTX 745. We achieve speedup of up to 10%. Input *fvcorr_097* has the smallest number of nodes, thus the smallest improvement. CFD already has an optimized data layout [9]. With our throttling method, we nonetheless see some speedup. This demonstrates the effectiveness of our approach.

6 Related Work

Modern GPUs are equipped with massive amounts of parallelism and significant computing horsepower. However, this also results in higher levels of cache contention. To achieve high performance, reusing data in cache is critical. Both software and hardware approaches have been proposed to address the cache contention problem.

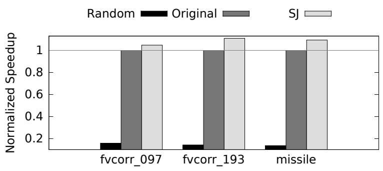


Figure 16: CFD Speedup

Warp Scheduling Policy Recent works focus on modifying GPU warp scheduling policy to reduce cache contention by throttling threads [18] [27] [20, 19] or to prioritize thread execution based on criticality [22]. However, all those approaches require hardware modification and require fine-grained thread scheduling which is complicated in a massively parallel system.

Our approach does not require hardware modification or fine-grained thread scheduling. Moreover, most warp scheduling policies aim to reduce the number of active warps for better performance. However, as we discovered in this paper, it is not always good to reduce the number of simultaneously running tasks for better cache performance. In some scenarios, i.e., when data reuse is low, having higher concurrency actually helps.

Computation and Data Layout Transformation On GPUs, Baskaran et al. [3] developed a compile-time transformation scheme coalescing loop nest accesses to achieve efficient global memory access. Zhang et al. [32] focused on reducing irregular memory accesses and enhancing memory coalescing to improve GPU program performance. These and other works [28, 31, 8, 17, 30, 4] all focus on improving memory coalescing for spatial locality. Our method is orthogonal to these approaches, as we optimize temporal locality.

7 Conclusion

This paper proposes a locality-aware software throttling framework that targets irregular sparse matrix applications on GPUs. We perform data-balanced *work partition* on the entire workload to get **cache-fit** partitions and use scheduling to exploit the trade-off between cache locality and execution pipeline utilization. Our framework is practical and effective. It requires no hardware modification and achieves an average 2.01X (maximal 6.45X) speedup on more than 200 real sparse matrices.

Acknowledgement We thank Peter Varman for being our shepherd and the anonymous reviewers for their constructive comments. This work is supported by grants NSF-CCF-1421505 and NSF-CCF-1628401. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Dimacs implementation challenge - shortest paths, July 2013.

[2] Akbudak, K., Kayaaslan, E., and Aykanat, C. Hypergraph partitioning based models and methods for exploiting cache locality in sparse matrix-vector multiplication. *SIAM J. Scientific Computing 35*, 3 (2013).

[3] Baskaran, M. M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd Annual International Conference on Supercomputing* (New York, NY, USA, 2008), ICS '08, ACM, pp. 225–234.

[4] Bell, N., and Garland, M. Efficient sparse matrix-vector multiplication on cuda. Tech. rep., Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[5] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517.

[6] Bourse, F., Lelarge, M., and Vojnovic, M. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2014), KDD '14, ACM, pp. 1456–1465.

[7] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)* (Washington, DC, USA, 2009), IISWC '09, IEEE Computer Society, pp. 44–54.

[8] Chen, X., Chang, L.-W., Rodrigues, C. I., Lv, J., Wang, Z., and Hwu, W.-M. Adaptive cache management for energy-efficient gpu computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2014), MICRO-47, IEEE Computer Society, pp. 343–355.

[9] Corrigan, A., Camelli, F., Löhner, R., and Wallin, J. Running unstructured grid cfd solvers on modern graphics hardware. In *19th AIAA Computational Fluid Dynamics Conference* (June 2009), no. AIAA 2009-4001.

[10] Dalton, S., and Bell, N. CUSP: A C++ templated sparse matrix library, 2014.

[11] Davis, T. A., and Hu, Y. The university of florida sparse matrix collection. *ACM Trans. Math. Softw. 38*, 1 (Dec. 2011), 1:1–1:25.

[12] Ding, C., and Kennedy, K. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1999), PLDI '99, ACM, pp. 229–241.

[13] Fredman, M. L. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing 5*, 1 (1976), 83–89.

[14] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), vol. 12, p. 2.

[15] Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)* (2016).

[16] Hoare, C. A. Algorithm 65: find. *Communications of the ACM 4*, 7 (1961), 321–322.

[17] Jang, B., Schaa, D., Mistry, P., and Kaeli, D. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Trans. Parallel Distrib. Syst. 22*, 1 (Jan. 2011), 105–118.

[18] Jog, A., Kayiran, O., Chidambaram Nachiappan, N., Mishra, A. K., Kandemir, M. T., Mutlu, O., Iyer, R., and Das, C. R. Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 395–406.

[19] Kayiran, O., Jog, A., Kandemir, M. T., and Das, C. R. Neither more nor less: Optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques* (Piscataway, NJ, USA, 2013), PACT '13, IEEE Press, pp. 157–166.

[20] Kayiran, O., Nachiappan, N. C., Jog, A., Ausavarungnirun, R., Kandemir, M. T., Loh, G. H., Mutlu, O., and Das, C. R. Managing gpu concurrency in heterogeneous architectures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2014), MICRO-47, IEEE Computer Society, pp. 114–126.

[21] Khorasani, F., Vora, K., Gupta, R., and Bhuyan, L. N. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing* (New York, NY, USA, 2014), HPDC '14, ACM, pp. 239–252.

[22] Lee, S.-Y., Arunkumar, A., and Wu, C.-J. Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 2015), ISCA '15, ACM, pp. 515–527.

[23] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[24] Li, A., Song, S. L., Liu, W., Liu, X., Kumar, A., and Corporaal, H. Locality-aware cta clustering for modern gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 297–311.

[25] Li, L., Geda, R., Hayes, A. B., Chen, Y., Chaudhari, P., Zhang, E. Z., and Szegedy, M. A simple yet effective balanced edge partition model for parallel computing. *Proc. ACM Meas. Anal. Comput. Syst. 1*, 1 (June 2017), 14:1–14:21.

[26] Li, L., Geda, R., Hayes, A. B., Chen, Y., Chaudhari, P., Zhang, E. Z., and Szegedy, M. A simple yet effective balanced edge partition model for parallel computing. In *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2017), SIGMETRICS '17 Abstracts, ACM, pp. 6–6.

[27] Rogers, T. G., O'Connor, M., and Aamodt, T. M. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2012), MICRO-45, IEEE Computer Society, pp. 72–83.

[28] Sung, I.-J., Stratton, J. A., and Hwu, W.-M. W. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 513–522.

[29] TSOURAKAKIS, C., GKANTSIDIS, C., RADUNOVIC, B., AND VOJNOVIC, M. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining* (New York, NY, USA, 2014), WSDM '14, ACM, pp. 333–342.

[30] VUDUC, R. W., AND MOON, H.-J. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *International Conference on High Performance Computing and Communications* (2005), Springer, pp. 807–816.

[31] YANG, Y., XIANG, P., KONG, J., AND ZHOU, H. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 86–97.

[32] ZHANG, E. Z., JIANG, Y., GUO, Z., AND SHEN, X. Streamlining gpu applications on the fly: Thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing* (New York, NY, USA, 2010), ICS '10, ACM, pp. 115–126.

[33] ZHANG, E. Z., JIANG, Y., GUO, Z., TIAN, K., AND SHEN, X. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 369–380.

Accelerating PageRank using Partition-Centric Processing

Kartik Lakhotia[1], Rajgopal Kannan[2], Viktor Prasanna[1]
[1]Ming Hsieh Department of Electrical Engineering, University of Southern California
[2]US Army Research Lab
[1]{klakhoti, prasanna}@usc.edu, [2]rajgopal.kannan.civ@mail.Mil

Abstract

PageRank is a fundamental link analysis algorithm that also functions as a key representative of the performance of Sparse Matrix-Vector (SpMV) multiplication. The traditional PageRank implementation generates fine granularity random memory accesses resulting in large amount of wasteful DRAM traffic and poor bandwidth utilization. In this paper, we present a novel Partition-Centric Processing Methodology (PCPM) to compute PageRank, that drastically reduces the amount of DRAM communication while achieving high sustained memory bandwidth. PCPM uses a Partition-centric abstraction coupled with the Gather-Apply-Scatter (GAS) programming model. By carefully examining how a PCPM based implementation impacts communication characteristics of the algorithm, we propose several system optimizations that improve the execution time substantially. More specifically, we develop (1) a new data layout that significantly reduces communication and random DRAM accesses, and (2) branch avoidance mechanisms to get rid of unpredictable data-dependent branches.

We perform detailed analytical and experimental evaluation of our approach using 6 large graphs and demonstrate an average 2.7× speedup in execution time and 1.7× reduction in communication volume, compared to the state-of-the-art. We also show that unlike other GAS based implementations, PCPM is able to further reduce main memory traffic by taking advantage of intelligent node labeling that enhances locality. Although we use PageRank as the target application in this paper, our approach can be applied to generic SpMV computation.

1 Introduction

Graphs are the preferred choice of data representation in many fields such as web and social network analysis [9, 3, 29, 10], biology [17], transportation [15, 4] etc. The growing scale of problems in these areas has generated substantial research interest in high performance graph analytics. A large fraction of this research is focused on shared memory platforms because of their low communication overhead compared to distributed systems [26]. High DRAM capacity in modern systems further allows in-memory processing of large graphs on a single server [35, 33, 37]. However, efficient utilization of compute power is challenging even on a single node because of the (1) low computation-to-communication ratio and, (2) irregular memory access patterns of graph algorithms. The growing disparity between CPU speed and DRAM bandwidth, termed memory wall [42], has become a key issue in high performance graph analytics.

PageRank is a quintessential algorithm that exemplifies the performance challenges posed by graph computations. It iteratively performs Sparse Matrix-Vector (SpMV) multiplication over the adjacency matrix of the target graph and the current PageRank vector $\overrightarrow{PR}$ to generate new PageRank values. The irregularity in adjacency matrices leads to random accesses to $\overrightarrow{PR}$ with poor *spatial* and *temporal* locality. The resulting cache misses and communication volume become the performance bottleneck for PageRank computation. Since many graph algorithms can be similarly modeled as a series of SpMV operations [37], optimizations on PageRank can be easily generalized to other algorithms.

Recent works have proposed the use of Gather-Apply-Scatter (GAS) model to improve locality and reduce communication for SpMV and PageRank [43, 11, 5]. This model splits computation into two phases: *scatter* current source node values on edges and *gather* propagated values on edges to compute new values for destination nodes. The 2-phased approach restricts access to either the current $\overrightarrow{PR}$ or new $\overrightarrow{PR}$ at a time. This provides opportunities for cache-efficient and lock-free parallelization of the algorithm.

We observe that although this approach exhibits several attractive features, it also has some drawbacks leading to inefficient memory accesses, both quantitative as well as qualitative. First, we note that while scattering, a vertex *repeatedly* writes its value on all outgoing edges, resulting in large number of reads and writes. We also observe that the Vertex-centric graph traversal in [11, 5] results in *random* DRAM accesses and the Edge-centric traversal in [34, 43] scans edge list in coordinate format which increases the number of reads.

Our premise is that by changing the focus of computation from a single vertex or edge to a *cacheable* group of vertices (partition), we can effectively identify and reduce redundant edge traversals as well as avoid random accesses to DRAM, while still retaining the benefits of GAS model. Based on these insights, we develop a new

Partition-Centric approach to compute PageRank. The major contributions of our work are:

1. We propose a Partition-Centric Processing Methodology (PCPM) that propagates updates from nodes to partitions and reduces the redundancy associated with GAS model.
2. By carefully evaluating how a PCPM based implementation impacts algorithm behavior, we develop several system optimizations that substantially accelerate the computation, namely, (a) a new data layout that drastically reduces communication and random memory accesses, (b) branch avoidance mechanisms to remove unpredictable branches.
3. We demonstrate that PCPM can take advantage of intelligent node labeling to further reduce the communication volume. Thus, PCPM is suitable even for high locality graphs.
4. We conduct extensive analytical and experimental evaluation of our approach using 6 large datasets. On a 16-core shared memory system, PCPM achieves $2.1\times - 3.8\times$ speedup in execution time and $1.3\times - 2.5\times$ reduction in main memory communication over state-of-the-art.
5. We show that PCPM can be easily extended to weighted graphs and generic SpMV computation (section 3.5) even though it is described in the context of PageRank algorithm in this paper.

2 Background and Related Work

2.1 PageRank Computation

In this section, we describe how PageRank is calculated and what makes it challenging for the conventional implementation to achieve high performance. Table 1 lists a set of notations that we use to mathematically represent the algorithm.

Table 1: List of graph notations

$G(V,E)$	Input directed graph
A	adjacency matrix of $G(V,E)$
$N_i(v)$	in-neighbors of vertex v
$N_o(v)$	out-neighbors of vertex v
$\overrightarrow{PR}_i$	PageRank value vector after i^{th} iteration
$\overrightarrow{SPR}$	scaled PageRank vector $\left(SPR(v) = \frac{PR_i(v)}{\lvert N_o(v)\rvert}\right)$
d	damping factor in PageRank algorithm

PageRank is computed iteratively. In each iteration, all vertex values are updated by the new weighted sum of their in-neighbors' PageRank, as shown in equation 1.

$$PR_{i+1}(v) = \frac{1-d}{|V|} + d \sum_{u \in N_i(v)} \frac{PR_i(u)}{\lvert N_o(u)\rvert} \tag{1}$$

PageRank is typically computed in pull direction [35, 38, 37, 30] where each vertex pulls the value of its in-neighbors and accumulates into its own value, as shown in algorithm 1. This corresponds to traversing A in a column-major order and computing the dot product of each column with the scaled PageRank vector $\overrightarrow{SPR}$.

Algorithm 1 Pull Direction PageRank (PDPR) Iteration

```
for $v \in V$ do
    $temp = 0$
    for all $u \in N_i(v)$ do
        $temp += PR[u]$
    $PR_{next}[v] = \frac{(1-d)\times|V|^{-1} + d\times temp}{|N_o(v)|}$
swap($PR, PR_{next}$)
```

In the pull direction implementation, each column completely owns the computation of the corresponding element in the output vector. This enables all columns of A to be traversed asynchronously in parallel without the need to store partial sums in memory. On the contrary, in the push direction, each node updates its out-neighbors by adding its own value to them. This requires a row-major traversal of A and storage for partial sums since each row contributes partially to multiple elements in the output vector. Further, synchronization is needed to ensure conflict-free processing of multiple rows that update the same output element.

Performance Challenges: Sparse matrix layouts like Compressed Sparse Column (CSC) store all non-zero elements of a column sequentially in memory allowing fast column-major traversal of A [36]. However, the neighbors of a node can be scattered anywhere in the graph and reading their values results in random accesses (single or double word) to $\overrightarrow{SPR}$ in pull direction computation. Similarly, the push direction implementation uses a Compressed Sparse Row (CSR) format for fast row-major traversal of A but suffers from random accesses to the partial sums vector. These low locality and fine granularity accesses incur high cache miss ratio and contribute a large fraction to the overall memory traffic as shown in fig. 1.

2.2 Related Work

The performance of PageRank depends heavily on the locality in memory access patterns of the graph (which we refer to as *graph locality*). Since node labeling has significant impact on graph locality, many prior works have investigated the use of node reordering or clustering [7, 22, 6, 2] to improve the performance of graph algorithms. Reordering based on spatial and temporal locality aware placement of neighbors [39, 20] has

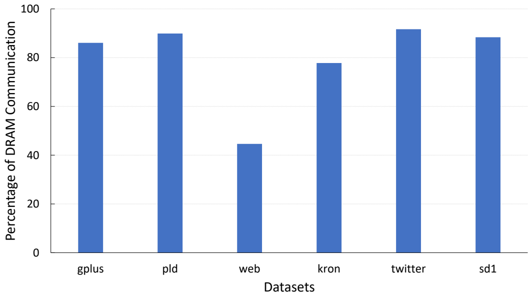


Figure 1: Percentage contribution of vertex value accesses to the total DRAM traffic in a PageRank iteration.

been shown to further outperform the well known clustering and tree-based techniques. However, such sophisticated algorithms also introduce substantial pre-processing overhead which limits their practicability. In addition, scale-free graphs like social networks are less tractable by reordering transformations because of their skewed degree distribution.

Cache Blocking (CB) is another technique used to accelerate graph processing [41, 32, 45]. CB induces locality by restricting the range of randomly accessed nodes and has been shown to reduce cache misses [24]. CB partitions A along rows, columns or both into multiple block matrices. However, SpMV computation with CB requires the partial sums to be re-read for each block. The extremely sparse nature of these block matrices also reduces the reuse of cached vertex data [31].

Gather-Apply-Scatter (GAS) is another popular model incorporated in many graph analytics frameworks [23, 34, 13]. It splits the analytic computation into *scatter* and *gather* phases. In the *scatter* phase, source vertices transmit updates on all of their outgoing edges and in the *gather* phase, these updates are processed to compute new values for corresponding destination vertices. The *updates* for PageRank algorithm correspond to *scaled PageRank values* defined earlier in section 2.1.

Binning exploits the 2-phased computation model by storing the updates in a semi-sorted manner. This induces *spatio-temporal* locality in access patterns of the algorithm. Binning can be used in conjunction with both Vertex-centric or Edge-centric paradigms. Zhou et al. [43, 44] use a custom sorted edge list with Edge-centric processing to reduce DRAM row activations and improve memory performance. However, their sorting mechanism introduces a non-trivial pre-processing cost and imposes the use of COO format. This results in larger communication volume and execution time than the CSR based Vertex-centric implementations [5, 11].

GAS model is also *inherently sub-optimal* when used with either Vertex-centric or Edge-centric abstractions. This is because it traverses the entire graph twice in each iteration. Nevertheless, Binning with Vertex-centric GAS (BVGAS) is the state-of-the-art methodology on shared memory platforms [5, 11] and we use it as baseline for comparison in this paper.

3 Partition-Centric Processing

We propose a new Partition-Centric Processing Methodology (PCPM) that significantly improves the efficiency of processor-memory communication over that achievable with current Vertex-centric or Edge-centric methods. We define *partitions* as disjoint sets of contiguously labeled nodes. The Partition-Centric abstraction then perceives the graph as a set of links from each node to the partitions corresponding to the neighbors of the node. We use this abstraction in conjunction with the 2-phased Gather-Apply-Scatter (GAS) model.

During the PCPM scatter phase, each thread processes one partition at a time. Processing a partition p means propagating messages from nodes in p to the neighboring partitions. A message to a partition p' comprises of the update value of source node ($PR[v]$) and the list of out-neighbors of v that lie in p'. PCPM caches the vertex data of p and streams the messages to the main memory. The messages from p are generated in a Partition-centric manner i.e. messages from all nodes in p to a neighboring partition p' are generated consecutively and are not interleaved with messages to any other partition.

During the gather phase, each thread scans all messages destined to one partition p at a time. A message scan applies the update value to all nodes in the neighbor list of that message. Partial sums of nodes in p are cached and messages are streamed from the main memory. After all messages to p are scanned, the partial sums (new PageRank values) are written back to DRAM.

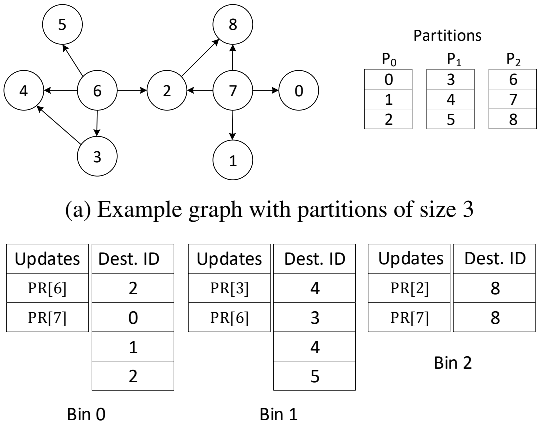
With static pre-allocation of distinct memory spaces for each partition to write messages, PCPM can asynchronously scatter or gather multiple partitions in parallel. In this section, we provide a detailed discussion on PCPM based computation and the required data layout.

3.1 Graph Partitioning

We employ a simple approach to divide the vertex set V into partitions. We create equisized partitions of size q where partition P_i owns all the vertices with index $\in [i*q,(i+1)*q)$ as shown in fig. 2a. As discussed later, the PCPM abstraction is built to easily take advantage of more sophisticated partitioning schemes and deliver further performance improvements (the trade-off is time complexity of partitioning versus performance gains). As we show in the results section, even the simple partitioning approach described above delivers significant performance gains over state-of-the-art methods.

Each partition is also allocated a contiguous memory space called *bin* to store updates (*update_bins*) and corresponding list of destination nodes (*destID_bins*) in the incoming messages. Since each thread in PCPM scatters or gathers only one partition at a time, the random accesses to vertex values or partial sums are limited to address range equal to the partition size. This improves temporal locality in access pattern and in turn, overall cache performance of the algorithm.

Before beginning PageRank computation, each partition calculates the offsets (address in bins where it must start writing from) into all *update_bins* and *destID_bins*. Our scattering strategy dictates that the partitions write to bins in the order of their IDs. Therefore, the offset for a partition P_i into any given bin is the sum of the number of values that all partitions with ID $< i$ are writing into that bin. For instance, in fig. 2, the offset of partition P_2 into *update_bins*[0] is 0 (since partitions P_0 and P_1 do not write to bin 0). Similarly, its offset into *update_bins*[1] and *update_bins*[2] is 1 (since P_1 writes one update to bin 1 and P_0 writes one update to bin 2). Offset computation provides each partition fixed and disjoint locations to write messages. This allows PCPM to parallelize partition processing without the need of locks or atomics.



(a) Example graph with partitions of size 3

(b) Bins store update value and list of destination nodes

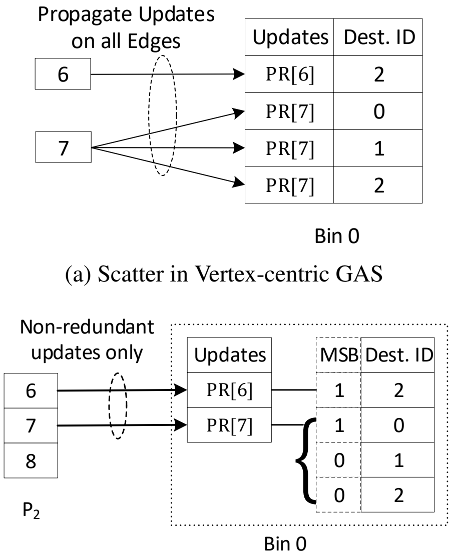
Figure 2: Graph Partitioning and messages inserted in bins during scatter phase

Note that since the destination node IDs written in the first iteration remain unchanged over the course of algorithm, they are written only once and reused in subsequent iterations. The reuse of destination node IDs along with the specific system optimizations discussed in section 3.2 and 3.3 enables PCPM to traverse only a fraction of the graph during scatter phase. This dramatically reduces the number of DRAM accesses and *gets rid of the inherent sub-optimality of GAS model.*

3.2 Partition-Centric Update Propagation

The unique abstraction of PCPM naturally leads to transmitting a single update from a node to a neighboring partition. In other words, even if a node has multiple neighbors in a partition, it inserts only one update value in the corresponding *update_bins* during scatter phase (algorithm 2). Fig. 3 illustrates the difference between Partition-Centric and Vertex-centric scatter for the example graph shown in fig. 2a.

PCPM manipulates the Most Significant Bit (MSB) of destination node IDs to indicate the range of nodes in a partition that use the same update value. In the *destID_bins*, it consecutively writes IDs of all nodes in the neighborhood of same source vertex and sets the MSB of first ID in this range to 1 for demarcation (fig. 3b). Since MSB is reserved for this functionality, PCPM supports graphs with upto 2 billion nodes instead of 4 billion for 4 Byte node IDs. However, to the best of our knowledge, this is enough to process most of the large publicly available datasets.



(a) Scatter in Vertex-centric GAS

(b) Scatter in PCPM

Figure 3: PCPM decouples *update_bins* and *destID_bins* to avoid redundant update value propagation

The gather phase starts only after all partitions are processed in the scatter phase. PCPM gather function sequentially reads updates and node IDs from the bins of the partition being processed. When gathering partition P_i, an update value $PR[v]$ should be applied to all out-neighbors of v that lie in P_i. This is done by checking the MSB of node IDs to determine whether to apply the previously read update or to read the next update, as shown in algorithm 2. The MSB is then masked to generate the true ID of destination node whose partial sum is updated.

Algorithm 2 describes PCPM based PageRank computation using a row-wise partitioned CSR format for adjacency matrix A. Note that PCPM only writes updates for *some* edges in a node's adjacency list, specifically the first outgoing edge to a partition. The remaining edges to that partition are *unused*. Since CSR stores adjacencies of a node contiguously, the set of first edges to neighboring partitions is interleaved with other edges. Therefore, we have to scan all outgoing edges of each vertex during scatter phase to access this set, which decreases efficiency. Moreover, the algorithm can potentially switch bins for each update insertion, leading to random writes to DRAM. Finally, the manipulation of MSB in node indices introduces additional data dependent branches which hurts the performance. Clearly, CSR adjacency matrix is not an efficient data layout for graph processing using PCPM. In the next section, we propose a PCPM-specific data layout.

Algorithm 2 PageRank iteration in PCPM using CSR format. Writing of *destID_bins* is not shown here.

```
$q \rightarrow$ partition size, $P \rightarrow$ set of partitions
for all $p \in P$ do in parallel                                  ▷ Scatter
    for all $v \in p$ do
        $prev\_bin \leftarrow \infty$
        for all $u \in N_o(v)$ do
            if $\lfloor u/q \rfloor \neq prev\_bin$ then
                insert $PR[v]$ in $update\_bins[\lfloor u/q \rfloor]$
                $prev\_bin \leftarrow \lfloor u/q \rfloor$
$PR[:] \leftarrow 0$
for all $p \in P$ do in parallel                                  ▷ Gather
    while $destID\_bins[p] \neq \emptyset$ do
        pop $id$ from $destID\_bins[p]$
        if $MSB(id) \neq 0$ then
            pop $update$ from $update\_bins[p]$
        $PR[id \ \& \ bitmask] \mathrel{+}= update$
for all $v \in V$ do in parallel                                  ▷ Apply
    $PR[v] \leftarrow \frac{(1-d)/|V| \ + \ d \times PR[v]}{|N_o(v)|}$
```

3.3 Data Layout Optimization

In this subsection, we describe a new bipartite Partition-Node Graph (PNG) data layout that brings out the true Partition-Centric nature of PCPM. During the scatter phase, PNG prevents unused edge reads and ensures that all updates to a bin are streamed together before switching to another bin.

We exploit the fact that once *destID_bins* are written, the only required information in PCPM is the connectivity between nodes and partitions. Therefore, edges going from a source to all destination nodes in a single partition can be *compressed* into one edge whose new destination is the corresponding partition number. This gives rise to a bipartite graph G' with disjoint vertex sets V and P (where $P = \{P_0, \ldots, P_{k-1}\}$ represents the set of partitions in the original graph), and a set of directed edges E' going from V to P. Such a transformation has the following effects:

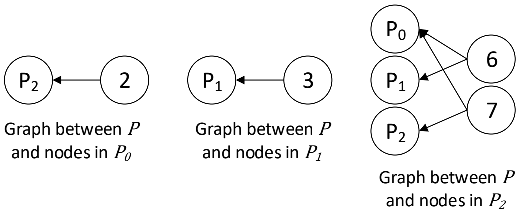

P2
2
Graph between P and nodes in P0
P1
3
Graph between P and nodes in P1
P0
P1
P2
6
7
Graph between P and nodes in P2

Figure 4: Partition-wise construction of PNG $G'(P,V,E')$ for graph $G(V,E)$ (fig. 2a). $|E'|$ is much smaller than $|E|$.

1. $Eff_1 \rightarrow$ the unused edges in original graph are removed
2. $Eff_2 \rightarrow$ the range of destination IDs reduces from $|V|$ to $|P|$.

The advantages of Eff_1 are obvious but those of Eff_2 will become clear when we discuss the storage format and construction of PNG.

The compression step reduces memory traffic by eliminating unused edge traversal. However note that scatters to a bin from source vertices in a partition are still interleaved with scatters to other bins. This can lead to random DRAM accesses during the scatter phase processing of a (source) partition. We resolve this problem by *transposing* the adjacency matrix of bipartite graph G'. The rows of the transposed matrix represent edges grouped by destination partitions which enables streaming updates to one bin at a time. This advantage comes at the cost of random accesses to source node values during the scatter phase. To prevent these random accesses from going to DRAM, we construct PNG on a per-partition basis i.e. we create a separate bipartite graph for each partition P_i with edges between P and the nodes in P_i (fig. 4). By carefully choosing q to make partitions *cacheable*, we ensure that all requests to source nodes are served by the cache resulting in *zero random DRAM accesses*.

Eff_2 is crucial for *transposition* of bipartite graphs in all partitions. The number of offsets required to store a transposed matrix in CSR format is equal to the range of destination node IDs. By reducing this range, Eff_2 reduces the storage requirement for offsets of each matrix from $O(|V|)$ to $O(|P|)$. Since there are $|P|$ partitions, each having one bipartite graph, the total storage requirement for edge offsets in PNG is $O(|P|^2)$ instead of $O(|V| \times |P|)$.

Although PNG construction looks like a 2-step approach, we actually merge *compression* and *transposition* into a single step. We first scan the outgoing edges of all nodes in a partition and individually compute the in-degree of all the destination partitions while discarding unused edges. A prefix sum of these degrees is carried out to compute the offsets array for CSR matrix. The same offsets can also be used to allocate disjoint writing locations into the bins of destination partitions. In the next scan, the edge array in CSR is filled with source node IDs completing both *compression* and *transposition*. PNG construction can be easily parallelized over all partitions to accelerate the pre-processing effectively.

Algorithm 3 shows the pseudocode for PCPM scatter phase using PNG layout. Unlike algorithm 2, the scatter function in algorithm 3 does not contain data dependent branches to check and discard unused edges. Using PNG provides drastic performance gains in PCPM scatter phase with little pre-processing overhead.

Algorithm 3 PCPM scatter phase using PNG layout

```
$G'(P,V,E') \rightarrow$ PNG, $N_i^p(p') \rightarrow$ in-neighbors of partition $p'$ in bipartite graph of partition $p$
**for all** $p \in P$ **do in parallel**                ▷ **Scatter**
    **for all** $p' \in P$ **do**
        **for all** $u \in N_i^p(p')$ **do**
            **insert** $PR[u]$ into $update\_bins[p']$
```

3.4 Branch Avoidance

Data dependent branches have been shown to have significant impact on performance of graph algorithms [14] and PNG removes such branches in PCPM scatter phase. In this subsection, we propose a branch avoidance mechanism for the PCPM gather phase. Branch avoidance enhances the sustained memory bandwidth but does not impact the amount of DRAM communication.

Note that the **pop** operations shown in algorithm 2 are implemented using pointers that increment after reading an entry from the respective bin. Let $destID\_ptr$ and $update\_ptr$ be the pointers to $destID\_bins[p]$ and $update\_bins[p]$, respectively. Note that the $destID\_ptr$ is incremented in every iteration whereas the $update\_ptr$ is only incremented if $MSB[id] \neq 0$.

To implement the branch avoiding gather function, instead of using a condition check over $MSB(id)$, we add it directly to $update\_ptr$. When $MSB(id)$ is 0, the pointer is not incremented and the same update value is read from cache in the next iteration; when $MSB(id)$ is 1, the pointer is incremented executing the **pop** operation on $update\_bins[p]$. The modified pseudocode for gather phase is shown in algorithm 4.

3.5 Weighted Graphs and SpMV

PCPM can be easily extended for computation on weighted graphs by storing the edge weights along with destination IDs in *destID_bins*. These weights can be read in the gather phase and applied to the source node value before updating the destination node. PCPM can also be extended to generic SpMV with non-square matrices by partitioning the rows and columns separately. In this case, the outermost loops in scatter phase (algorithm 3) and gather phase (algorithm 4) will iterate over row partitions and column partitions of A, respectively.

Algorithm 4 Branch Avoiding gather function in PCPM

```
$PR[:] = 0$
**for all** $p \in P$ **do in parallel**                ▷ **Gather**
    $\{destID\_ptr,\ update\_ptr\} \leftarrow 0$
    **while** $destID\_ptr < size(destID\_bins[p])$ **do**
        $id \leftarrow destID\_bins[p][destID\_ptr{+}{+}]$
        $update\_ptr \mathrel{+}= MSB(id)$
        $id \leftarrow id\ \&\ bitmask$
        $PR[id] \mathrel{+}= update\_bins[p][update\_ptr]$
```

4 Comparison with Vertex-centric GAS

The Binning with Vertex-centric GAS (BVGAS) method allocates multiple bins to store incoming messages ($(update, destID)$ pairs). If bin width is q, then all messages destined to $v \in [i*q, (i+1)*q)$ are written in bin i. The scatter phase traverses the graph in a Vertex-centric fashion and inserts the messages in respective bins of the destination vertices. Number of bins is kept small to allow insertion points for all bins to fit in cache, providing good spatial locality. The gather phase processes one bin at a time as shown in algorithm 5, and thus, enjoys good temporal locality if bin width is small.

Algorithm 5 PageRank Iteration using BVGAS

```
$q \rightarrow$ bin width, $B \rightarrow$ no. of bins
**for** $v \in V$ **do**                              ▷ **Scatter**
    $PR[v] = PR[v]/|N_o(v)|$
    **for all** $u \in N_o(v)$ **do**
        **insert** $(PR[v], u)$ into $bins[\lfloor u/q \rfloor]$
$PR[:] = 0$
**for** $b = 0$ to $B-1$ **do**                       ▷ **Gather**
    **for all** $(update, dest)$ in $bins[b]$ **do**
        $PR[dest] = PR[dest] + update$
**for all** $v \in V$ **do**                          ▷ **Apply**
    $PR[v] = \frac{(1-d)}{|V|} + d \times PR[v]$
```

Unlike algorithm 5, in our BVGAS implementation, we write the destination IDs only in the first iteration. We also use small cached buffers to store updates before writing to DRAM. This ensures full cache line utilization and reduces communication during scatter phase [5].

Irrespective of all the locality advantages and optimizations, BVGAS inherently suffers from redundant reads and writes of a vertex value on all of its outgoing

Table 2: List of model parameters

Original Graph $G(V,E)$		PNG layout $G'(P,V,E')$	
n	no. of vertices ($\lvert V \rvert$)	k	no. of partitions ($\lvert P \rvert$)
m	no. of edges ($\lvert E \rvert$)	r	compression ratio ($\lvert E \rvert / \lvert E' \rvert$)

Architecture		Software	
c_{mr}	cache miss ratio for source value reads in PDPR	d_v	sizeof (updates/PageRank value)
l	sizeof (cache line)	d_i	sizeof (node or edge index)

edges. This redundancy manifests itself in the form of BVGAS' inability to utilize *high locality* in graphs with optimized node labeling. PCPM on the other hand, uses graph locality to reduce the fraction of graph traversed in scatter phase. Unlike PCPM, the Vertex-centric traversal in BVGAS can also insert consecutive updates into different bins. This leads to random DRAM accesses and poor bandwidth utilization. We provide a quantitative analysis of these differences in the next section.

5 Analytical Evaluation

We derive performance models to compare PCPM against conventional Pull Direction PageRank (PDPR) and BVGAS. Our models provide valuable insights into the behavior of different methodologies with respect to varying graph structure and locality. Table 2 defines the parameters used in the analysis. We use a synthetic kronecker graph [28] of scale 25 *(kron)* as an example for illustration purposes.

5.1 DRAM Communication

We analyze the amount of data exchanged with main memory per iteration of PageRank. We assume that data is accessed in quantum of one cache line and BVGAS exhibits full cache line utilization. Since destination indices are written only in the first iteration for PCPM and BVGAS, they are not accounted for in this model.

PDPR: The pull technique scans all edges in the graph once (algorithm 1). For a CSR format, this requires reading n edge offsets and m source node indices. PDPR also reads m source node values that incur cache misses generating $mc_{mr}l$ Bytes of DRAM traffic. Outputting new PageRank values generates nd_v Bytes of writes to DRAM. The total communication volume for PDPR is:

$$PDPR_{comm} = m(d_i + c_{mr}l) + n(d_i + d_v) \tag{2}$$

BVGAS: The scatter phase (algorithm 5) scans the graph and writes updates on all outgoing edges of the source node, thus communicating $(n+m)d_i + (n+m)d_v$ Bytes. The gather phase loads updates and destination node IDs on all the edges generating $m(d_i + d_v)$ Bytes of read traffic. At the end of gather phase, nd_v Bytes of new PageRank values are written in the main memory. Total communication volume for BVGAS is therefore, given by:

$$BVGAS_{comm} = 2m(d_i + d_v) + n(d_i + 2d_v) \tag{3}$$

PCPM with PNG: Number of edge offsets in bipartite graph of each partition is k. Thus, in the scatter phase (algorithm 3), a scan of PNG reads $(k \times k + {}^{m}\!/_{r})d_i$ Bytes. The scatter phase further reads n PageRank values and writes updates on ${}^{m}\!/_{r}$ edges. The gather phase (algorithm 4) reads m destination IDs and ${}^{m}\!/_{r}$ updates followed by n new PageRank value writes. Net communication volume in PCPM is given by:

$$PCPM_{comm} = m\left(d_i\left(1 + \frac{1}{r}\right) + \frac{2d_v}{r}\right) + k^2 d_i + 2nd_v \tag{4}$$

Comparison: Performance of pull technique depends heavily on c_{mr}. In the worst case, all accesses are cache misses i.e. $c_{mr} = 1$ and in best case, only cold misses are encountered to load the PageRank values in cache i.e. $c_{mr} = {}^{nd_v}\!/_{ml}$. Assuming $k^2 \ll n \ll m$, we get $PDPR_{comm} \in [md_i, m(d_i + l)]$. On the other hand, communication for BVGAS stays constant. With $\theta(m)$ additional loads and stores, $BVGAS_{comm}$ can never reach the lower bound of $PDPR_{comm}$. Comparatively, $PCPM_{comm}$ achieves optimality when for every vertex, all outgoing edges can be compressed into a single edge i.e. $r = {}^{m}\!/_{n}$. In the worst case when $r = 1$, PCPM is still as good as BVGAS and we get $PCPM_{comm} \in [md_i, m(2d_i + 2d_v)]$. Unlike BVGAS, $PCPM_{comm}$ achieves the same lower bound as $PDPR_{comm}$.

Analyzing equations 2 and 3, we see that BVGAS is profitable compared to PDPR when:

$$c_{mr} > \frac{d_i + 2d_v}{l} \tag{5}$$

In comparison, PCPM offers a more relaxed constraint on c_{mr} (by a factor of ${}^{1}\!/_{r}$) becoming advantageous when:

$$c_{mr} > \frac{d_i + 2d_v}{rl} \tag{6}$$

The RHS in eq. 5 is constant indicating that BVGAS is advantageous for low locality graphs. With optimized node ordering, we can reduce c_{mr} and outperform BVGAS. On the contrary, $r \in [1, {}^{m}\!/_{n}]$ in the RHS of eq. 6 is a function of locality. With an optimized node labeling, r also increases and enhances the performance of PCPM. Fig. 5 shows the effect of r on predicted DRAM communication for the *kron* graph. Obtaining an optimal nodel labeling that makes $r = {}^{m}\!/_{n}$ might be very difficult or even impossible for some graphs. However, as can be observed from fig. 5, DRAM traffic decreases rapidly for $r \leq 5$ and converges slowly for $r > 5$. Therefore, a node reordering that can achieve $r \approx 5$ is good enough to optimize communication volume in PCPM.

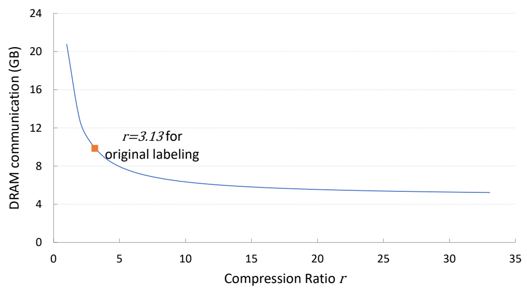


Figure 5: Predicted DRAM traffic for *kron* graph with $n = 33.5$ M, $m = 1070$ M, $k = 512$ and $d_i = d_v = 4$ Bytes.

5.2 Random Memory Accesses

We define a random access as a non-sequential jump in the address of memory location being read from or written to DRAM. Random accesses can incur latency penalties and negatively impact the sustained memory bandwidth. In this subsection, we model the amount of random accesses performed by different methodologies in a single PageRank iteration.

PDPR: Reading edge offsets and source node IDs in pull technique is completely sequential because of the CSR format. However, all accesses to source node PageRank values served by DRAM contribute to potential random accesses resulting in:

$$PDPR_{ra} = O(mc_{mr}) \tag{7}$$

BVGAS: In scatter phase of algorithm 5, updates can potentially be inserted at random memory locations. Assuming full cache line utilization for BVGAS, for every l Bytes written, there is at most 1 random DRAM access. In gather phase, all DRAM accesses are sequential if we assume that bin width is smaller than the cache. Total random accesses for BVGAS are then given by:

$$BVGAS_{ra} = O\left(\frac{md_v}{l}\right) \tag{8}$$

PCPM: With the PNG layout (algorithm 3), there are at most k bin switches when scattering updates from a partition. Since there are k such partitions, total number of random accesses in PCPM is bound by:

$$PCPM_{ra} = O(k * k) = O(k^2) \tag{9}$$

Comparison: BVGAS exhibits less random accesses than PDPR. However, $PCPM_{ra}$ is much smaller than both $BVGAS_{ra}$ and $PDPR_{ra}$. For instance, in the *kron* dataset with $d_v = 4$ Bytes, $l = 64$ Bytes and $k = 512$, $BVGAS_{ra} \approx 66.9$ M whereas $PCPM_{ra} \approx 0.26$ M.

Although it is not indicated in algorithm 5, the number of data dependent unpredictable branches in cache bypassing BVGAS implementation is also $O(m)$. For every update insertion, the BVGAS scatter function has to check if the corresponding cached buffer is full (section 4). In contrast, the number of branch mispredictions for PCPM (using branch avoidance) is $O(k^2)$ with 1 misprediction for every destination partition (p') switch in algorithm 3. The derivations are similar to random access model and for the sake of brevity, we do not provide a detailed deduction.

6 Experimental Evaluation

6.1 Experimental Setup and Datasets

We conduct experiments on a dual-socket Ivy Bridge server equipped with two 8-core Intel Xeon E5-2650 v2 processors@2.6 GHz running Ubuntu 14.04 OS. Table 3 lists important characteristics of our machine. Memory bandwidth is measured using STREAM benchmark [25]. All codes are written in C++ and compiled using G++ 4.7.1 with the highest optimization -O3 flag. The memory statistics are collected using Intel Performance Counter Monitor [40]. All data types used for indices and PageRank values are 4 Bytes.

Table 3: System Characteristics

Socket	no. of cores	8
	shared L3 cache	25MB
Core	L1d cache	32 KB
	L2 cache	256 KB
Memory	size	128 GB
	Read BW	59.6 GB/s
	Write BW	32.9 GB/s

We use 6 large real world and synthetic graph datasets coming from different applications, for performance evaluation. Table 4 summarizes the size and sparsity characteristics of these graphs. *Gplus* and *twitter* are follower graphs on social networks; *pld*, *web* and *sd1* are hyperlink graphs obtained by web crawlers; and *kron* is a scale 25 graph generated using Graph500 Kronecker generator. The *web* is a very sparse graph but has high locality obtained by a very expensive pre-processing of node labels [6]. The *kron* graph has higher edge density as compared to other datasets.

Table 4: Graph Datasets

Dataset	Description	# Nodes (M)	# Edges (M)	Degree
gplus [12]	Google Plus	28.94	462.99	16
pld [27]	Pay-Level-Domain	42.89	623.06	14.53
web [6]	Webbase-2001	118.14	992.84	8.4
kron [28]	Synthetic graph	33.5	1047.93	31.28
twitter [19]	Follower network	61.58	1468.36	23.84
sd1 [27]	Subdomain graph	94.95	1937.49	20.4

6.2 Implementation Details

We use a simple hand coded implementation of algorithm 1 for PDPR and parallelize it over vertices with static load balancing on the number of edges traversed. Our baseline does not incur overheads associated with similar implementations in frameworks [35, 30, 37] and hence, is faster than framework based programs [5].

To parallelize BVGAS scatter phase (algorithm 5), we give each thread a fixed range of nodes to scatter. Work per thread is statically balanced in terms of the number of edges processed. We also give each thread distinct memory spaces corresponding to all bins to avoid atomicity concerns in scatter phase. We use the Intel AVX non-temporal store instructions [1] to bypass the cache while writing updates and use 128 Bytes cache line aligned buffers to accumulate the updates for streaming stores [5]. BVGAS gather phase is parallelized over bins with load balanced using OpenMP dynamic scheduling. The optimal bin width is empirically determined and set to 256 KB (64K nodes). As bin width is a power of 2, we use bit shift instructions instead of integer division to compute the destination bin from node ID.

The PCPM scatter and gather phases are parallelized over partitions and load balancing in both the phases is done dynamically using OpenMP. Partition size is empirically determined and set to 256 KB. A detailed design space exploration of PCPM is discussed in section 6.3.2.

All the implementations mentioned in this section execute 20 PageRank iterations on 16 cores. For accuracy of the collected information, we repeat these algorithms 5 times and report the average values.

6.3 Results

6.3.1 Comparison with Baselines

Execution Time: Fig. 6 gives a comparison of the GTEPS (computed as the ratio of giga edges in the graph to the runtime of single PageRank iteration) achieved by different implementations. We observe that PCPM is $2.1-3.8\times$ faster than the state-of-the-art BVGAS implementation and upto $4.1\times$ faster than PDPR. BVGAS achieves constant throughput irrespective of the graph structure and is able to accelerate computation on low locality graphs. However, it is worse than PDPR for high locality (*web*) and dense (*kron*) graphs. PCPM is able to outperform PDPR and BVGAS on all datasets, though the speedup on *web* graph is minute because of high performance of PDPR. Detailed results for execution time of BVGAS and PCPM during different phases of computation are given in table 5. PCPM scatter phase benefits from a multitude of optimizations to achieve a dramatic $5\times$ speedup over BVGAS scatter phase.

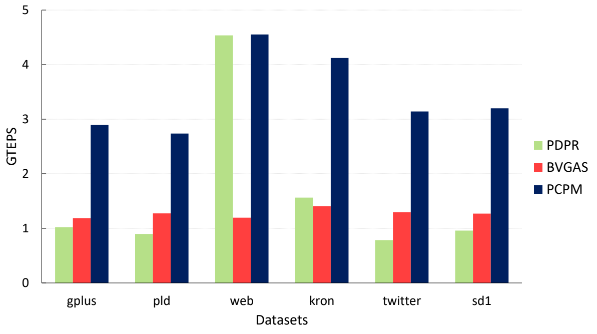


Figure 6: Performance in GTEPS. PCPM provides substantial speedup over BVGAS and PDPR.

Table 5: Execution time per iteration of PageRank for PDPR, BVGAS and PCPM

	PDPR	BVGAS			PCPM		
Dataset	Total Time(s)	Scatter Time(s)	Gather Time(s)	Total Time(s)	Scatter Time(s)	Gather Time(s)	Total Time(s)
gplus	0.44	0.26	0.12	0.38	0.06	0.1	0.16
pld	0.68	0.33	0.15	0.48	0.09	0.13	0.22
web	0.21	0.58	0.23	0.81	0.04	0.17	0.21
kron	0.65	0.5	0.22	0.72	0.07	0.18	0.25
twitter	1.83	0.79	0.32	1.11	0.18	0.27	0.45
sd1	1.97	1.07	0.42	1.49	0.24	0.35	0.59

Communication and Bandwidth: Fig. 7 shows the amount of data communicated with main memory normalized by the number of edges in the graph. Average communication in PCPM is $1.7\times$ and $2.2\times$ less than BVGAS and PDPR, respectively. Further, PCPM memory traffic per edge for *web* and *kron* is lower than other graphs because of their high compression ratio (table 6). The normalized communication for BVGAS is almost constant and therefore, its utility depends on the efficiency of pull direction baseline.

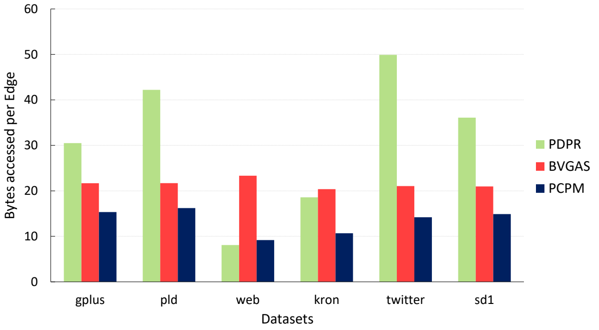


Figure 7: Main memory traffic per edge. PCPM communicates the least for all datasets except the *web* graph.

Note that the speedup obtained by PCPM is larger than the reduction in communication volume. This is because by avoiding random DRAM accesses and unpredictable branches, PCPM is able to efficiently utilize the available DRAM bandwidth. As shown in fig. 8, PCPM can sustain an average 42.4 GB/s bandwidth compared to 33.1 GB/s and 26 GB/s of PDPR and BVGAS, respectively. For large graphs like *sd1*, PCPM achieves $\approx 77\%$ of the peak read bandwidth (table 3) of our system. Although both PDPR and BVGAS suffer from random memory accesses, the former executes very few instructions and therefore, has better bandwidth utilization.

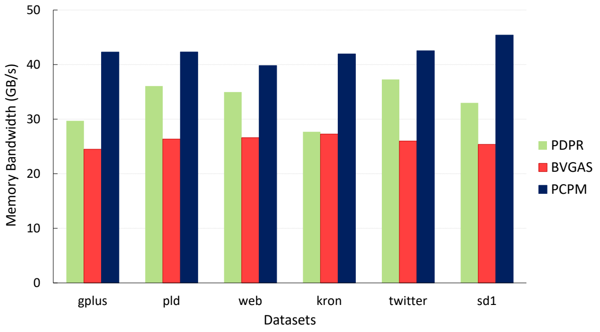


Figure 8: Sustained Memory Bandwidth for different methods. PCPM achieves highest bandwidth utilization.

Table 6: Locality vs compression ratio r. GOrder improves locality in neighbors and increases compression

		Original Labeling		GOrder Labeling	
Dataset	#Edges in Graph (M)	#Edges in PNG (M)	r	#Edges in PNG (M)	r
gplus	463	243.8	1.9	157.4	2.94
pld	623.1	347.7	1.79	166.7	3.73
web	992.8	118.1	8.4	126.8	7.83
kron	104.8	342.7	3.06	169.7	6.17
twitter	1468.4	722.4	2.03	386.2	3.8
sd1	1937.5	976.9	1.98	366.2	5.29

The reduced communication and streaming access patterns in PCPM also enhance its energy efficiency resulting in lower μJ/edge consumption as compared to BVGAS and PDPR, as shown in fig. 9. Energy efficiency is important from an eco-friendly computing perspective as highlighted by the Green Graph500 benchmark [16].

Effect of Locality: To assess the impact of locality on different methodologies, we relabel the nodes in our graph datasets using the GOrder [39] algorithm. We refer to the original node labeling in graph as *Orig* and GOrder labeling as simply *GOrder*. *GOrder* increases spatial locality by placing nodes with common in-neighbors closer in the memory. As a result, outgoing edges of the nodes tend to be concentrated in few partitions which increases the compression ratio r as shown in table 6. However, the *web* graph exhibits near optimal compression ($r = 8.4$) with *Orig* and does not show improvement with *GOrder*.

Table 7 shows the impact of *GOrder* on DRAM communication. As expected, BVGAS communicates a constant amount of data for a given graph irrespective of the labeling scheme used. On the contrary, memory traffic generated by PDPR and PCPM decreases because of reduced c_{mr} and increased r, respectively. These observations are in complete accordance with the performance models discussed in section 5.1. The effect on PCPM is not as drastic as PDPR because after r becomes greater than a threshold, PCPM communication decreases slowly as shown in fig. 5. Nevertheless, for almost all of the datasets, the net data transferred in PCPM is remarkably lesser than both PDPR and BVGAS for either of the vertex labelings.

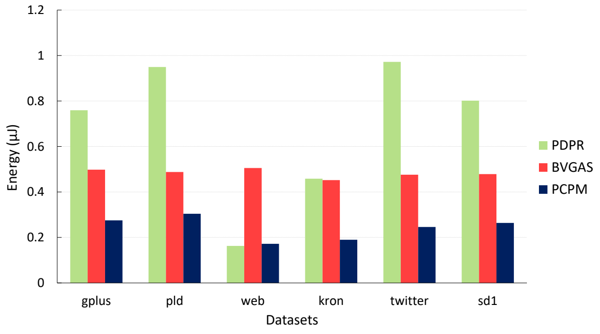


Figure 9: DRAM energy consumption per edge. PCPM benefits from reduced communication and random memory accesses.

Table 7: DRAM data transfer per iteration (in GB). PDPR and PCPM benefit from optimized node labeling

	PDPR		BVGAS		PCPM	
Dataset	*Orig*	*GOrder*	*Orig*	*GOrder*	*Orig*	*GOrder*
gplus	13.1	7.4	9.3	9.3	6.6	5.1
pld	24.5	10.7	12.6	12.5	9.4	6.1
web	7.5	7.6	21.6	21.3	8.5	8.4
kron	18.1	10.8	19.9	19.5	10.4	7.5
twitter	68.2	31.6	28.8	28.2	19.4	13.4
sd1	65.1	23.8	37.8	37.8	26.9	15.6

6.3.2 PCPM Design Space Exploration

Partition size represents an important tradeoff in PCPM. Large partitions force neighbors of each node to fit in fewer partitions resulting in better compression but poor locality. Small partitions on the other hand ensure high locality random accesses within partitions but reduce compression. We evaluate the impact of partition size on the performance of PCPM by varying it from 32 KB (8K nodes) to 8 MB (2M nodes). We observe a reduction in DRAM communication volume with increasing partition size (fig. 10). However, increases partition size beyond what cache can accommodate results in cache misses and a drastic increase in the DRAM traffic. As an exception, the performance on *web* graph is not heavily affected by partition size because of its high locality.

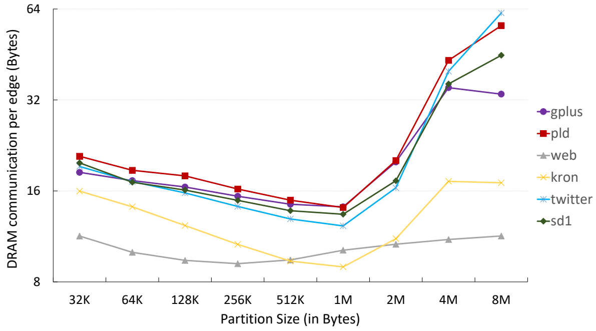


Figure 10: Impact of partition size on communication volume. Very large partitions result in cache misses and increased DRAM traffic.

The execution time (fig. 11) also benefits from communication reduction and is penalized by cache misses for large partitions. Note that for partition sizes > 256 KB and <= 1 MB, communication volume decreases but execution time increases. This is because in this range, many requests are served from the larger shared L3 cache which is slower than the private L1 and L2 caches. This phenomenon decelerates the computation but does not add to DRAM traffic.

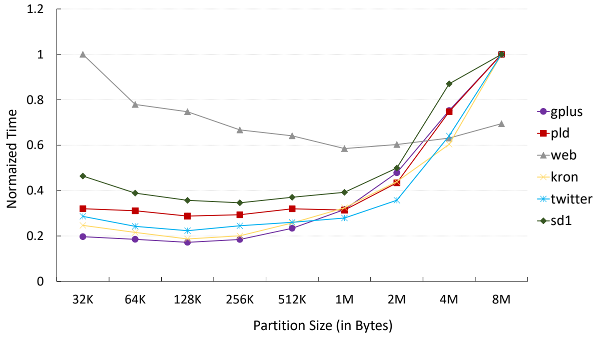


Figure 11: Impact of partition size on execution time.

Table 8: Pre-processing time of different methodologies. PNG construction increases the overhead of PCPM

Dataset	PCPM	BVGAS	PDPR
gplus	0.25s	0.1s	0s
pld	0.32s	0.15s	0s
web	0.26s	0.18s	0s
kron	0.43s	0.22s	0s
twitter	0.7s	0.27s	0s
sd1	0.95s	0.32s	0s

6.3.3 Pre-processing Time

We assume that adjacency matrix in CSR and CSC format is available and hence, PDPR does not need any pre-processing. Both BVGAS and PCPM however, require a beforehand computation of bin size and write offsets incurring non-zero pre-processing time as shown in table 8. In addition, PCPM also constructs the PNG layout. Fortunately, the computation of write offsets can be easily merged with PNG construction (section 3.3) to reduce the overhead. The pre-processing time also gets amortized over multiple iterations of PageRank.

7 Conclusion and Future Work

In this paper, we formulated a Partition-Centric Processing Methodology (PCPM) that perceives a graph as a set of links between nodes and partitions instead of nodes and their individual neighbors. We presented several features of this abstraction and developed data layout and system level optimizations to exploit them.

We conducted extensive analytical and experimental evaluation of our approach. Using a simple index based partitioning, we observed an average 2.7× speedup in execution time and 1.7× reduction in DRAM communication volume over state-of-the-art. In the future, we will explore edge partitioning models [21, 8] to further reduce communication and improve load balancing for PCPM.

Although we demonstrate the advantages of PCPM on PageRank, we show that it can be easily extended to generic SpMV computation. We believe that PCPM can be an efficient programming model for other graph algorithms or graph analytics frameworks. In this context, there are many promising directions for further exploration. For instance, the streaming memory access patterns of PNG enabled PCPM are highly suitable for High Bandwidth Memory (HBM) and disk-based systems. Exploring PCPM as a programming model for heterogenous memory or processor architectures is an interesting avenue for future work.

PCPM accesses nodes from only one graph partition at a time. Hence, G-Store's smallest number of bits representation [18] can be used to reduce the memory footprint and DRAM communication even further. Devising novel methods for enhanced compression can also make PCPM amenable to be used for large-scale graph processing on commodity PCs.

Acknowledgements: This material is based on work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086, National Science Foundation (NSF) under Contract Numbers CNS-1643351 and ACI-1339756 and Air Force Research Laboratory under Grant Number FA8750-15-1-0185. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, NSF or AFRL. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] Intel c++ compiler 17.0 developer guide and reference, 2016. Available at `https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-reference-guide`.

[2] Abou-Rjeili, A., and Karypis, G. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing* (2006), IPDPS'06, IEEE Computer Society, pp. 124–124.

[3] Albert, R., Jeong, H., and Barabási, A.-L. Internet: Diameter of the world-wide web. *nature 401*, 6749 (1999), 130.

[4] Aldous, J. M., and Wilson, R. J. *Graphs and applications: an introductory approach*, vol. 1. Springer Science & Business Media, 2003.

[5] Beamer, S., Asanović, K., and Patterson, D. Reducing pagerank communication via propagation blocking. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International* (2017), IEEE, pp. 820–831.

[6] Boldi, P., Rosa, M., Santini, M., and Vigna, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web* (2011), ACM, pp. 587–596.

[7] Boldi, P., Santini, M., and Vigna, S. Permuting web and social graphs. *Internet Mathematics 6*, 3 (2009), 257–283.

[8] Bourse, F., Lelarge, M., and Vojnovic, M. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2014), KDD '14, ACM, pp. 1456–1465.

[9] Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., and Wiener, J. Graph structure in the web. *Computer networks 33*, 1-6 (2000), 309–320.

[10] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H. C., et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference* (2013), pp. 49–60.

[11] Buono, D., Petrini, F., Checconi, F., Liu, X., Que, X., Long, C., and Tuan, T.-C. Optimizing sparse matrix-vector multiplication for large-scale data analytics. In *Proceedings of the 2016 International Conference on Supercomputing* (2016), ACM, p. 37.

[12] Gong, N. Z., Xu, W., Huang, L., Mittal, P., Stefanov, E., Sekar, V., and Song, D. Evolution of social-attribute networks: measurements, modeling, and implications using google+. In *Proceedings of the 2012 Internet Measurement Conference* (2012), ACM, pp. 131–144.

[13] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012), USENIX, pp. 17–30.

[14] Green, O., Dukhan, M., and Vuduc, R. Branch-avoiding graph algorithms. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures* (2015), ACM, pp. 212–223.

[15] Haklay, M., and Weber, P. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing 7*, 4 (2008), 12–18.

[16] Hoefler, T. Green graph500. Available at `http://green.graph500.org/`.

[17] Huber, W., Carey, V. J., Long, L., Falcon, S., and Gentleman, R. Graphs in molecular biology. *BMC bioinformatics 8*, 6 (2007), S8.

[18] Kumar, P., and Huang, H. H. G-store: high-performance graph store for trillion-edge processing. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for* (2016), IEEE, pp. 830–841.

[19] Kwak, H., Lee, C., Park, H., and Moon, S. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web* (2010), ACM, pp. 591–600.

[20] Lakhotia, K., Singapura, S., Kannan, R., and Prasanna, V. Recall: Reordered cache aware locality based graph processing. In *High Performance Computing (HiPC), 2017 IEEE 24th International Conference on* (2017), IEEE, pp. 273–282.

[21] Li, L., Geda, R., Hayes, A. B., Chen, Y., Chaudhari, P., Zhang, E. Z., and Szegedy, M. A simple yet effective balanced edge partition model for parallel computing. In *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems* (2017), SIGMETRICS '17 Abstracts, ACM, pp. 6–6.

[22] Liu, W.-H., and Sherman, A. H. Comparative analysis of the cuthill–mckee and the reverse cuthill–mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis 13*, 2 (1976), 198–213.

[23] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 135–146.

[24] Malicevic, J., Lepers, B., and Zwaenepoel, W. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), USENIX, pp. 631–643.

[25] McCalpin, J. D. Stream benchmark. *Link: www. cs. virginia. edu/stream/ref. html# what 22* (1995).

[26] McSherry, F., Isard, M., and Murray, D. G. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (2015), HOTOS'15, USENIX Association, pp. 14–14.

[27] Meusel, R., Vigna, S., Lehmberg, O., and Bizer, C. The graph structure in the web: Analyzed on different aggregation levels. *The Journal of Web Science 1*, 1 (2015), 33–47.

[28] Murphy, R. C., Wheeler, K. B., Barrett, B. W., and Ang, J. A. Introducing the graph 500. *Cray Users Group (CUG) 19* (2010), 45–74.

[29] Newman, M. E., Watts, D. J., and Strogatz, S. H. Random graph models of social networks. *Proceedings of the National Academy of Sciences 99*, suppl 1 (2002), 2566–2572.

[30] Nguyen, D., Lenharth, A., and Pingali, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 456–471.

[31] Nishtala, R., Vuduc, R. W., Demmel, J. W., and Yelick, K. A. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing 18*, 3 (2007), 297–311.

[32] Penner, M., and Prasanna, V. K. Cache-friendly implementations of transitive closure. *Journal of Experimental Algorithmics (JEA) 11* (2007), 1–3.

[33] Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., and Haradasan, M. Managing large graphs on multi-cores with graph awareness. 41–52.

[34] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 472–488.

[35] Shun, J., and Blelloch, G. E. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices* (2013), vol. 48, ACM, pp. 135–146.

[36] Siek, J. G., Lee, L.-Q., and Lumsdaine, A. *The Boost Graph Library: User Guide and Reference Manual, Portable Documents*. Pearson Education, 2001.

[37] Sundaram, N., Satish, N., Patwary, M. M. A., Dulloor, S. R., Anderson, M. J., Vadlamudi, S. G., Das, D., and Dubey, P. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1214–1225.

[38] Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices* (2016), vol. 51, ACM, p. 11.

[39] Wei, H., Yu, J. X., Lu, C., and Lin, X. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data* (2016), ACM, pp. 1813–1828.

[40] Willhalm, T., Dementiev, R., and Fay, P. Intel performance counter monitor-a better way to measure cpu utilization. 2012. *URL: http://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpuutilization* (2016).

[41] WILLIAMS, S., OLIKER, L., VUDUC, R., SHALF, J., YELICK, K., AND DEMMEL, J. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing 35*, 3 (2009), 178–194.

[42] WULF, W. A., AND MCKEE, S. A. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news 23*, 1 (1995), 20–24.

[43] ZHOU, S., CHELMIS, C., AND PRASANNA, V. K. Optimizing memory performance for fpga implementation of pagerank. In *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on* (2015), IEEE, pp. 1–6.

[44] ZHOU, S., LAKHOTIA, K., SINGAPURA, S. G., ZENG, H., KANNAN, R., PRASANNA, V. K., FOX, J., KIM, E., GREEN, O., AND BADER, D. A. Design and implementation of parallel pagerank on multicore platforms. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE* (2017), IEEE, pp. 1–6.

[45] ZHU, X., HAN, W., AND CHEN, W. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), USENIX Association, pp. 375–386.

CGraph: A Correlations-aware Approach for Efficient Concurrent Iterative Graph Processing

Yu Zhang[†] *Xiaofei Liao*[†*] *Hai Jin*[†] *Lin Gu*[†] *Ligang He*[§] *Bingsheng He*[‡] *Haikun Liu*[†]
[†] *Services Computing Technology and System Lab,*
Big Data Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, China

[§]*Department of Computer Science, University of Warwick, UK*

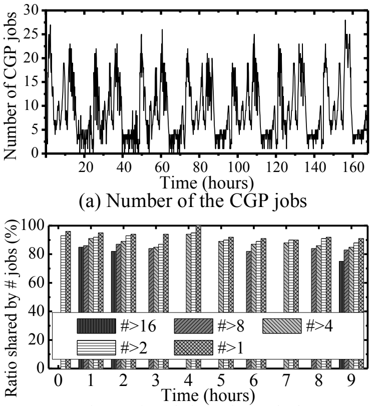
[‡]*Department of Computer Science, National University of Singapore, Singapore*
{zhyu, xfliao, hjin, lingu, hkliu}@hust.edu.cn ligang.he@warwick.ac.uk hebs@comp.nus.edu.sg

Abstract

With the fast growing of iterative graph analysis applications, the graph processing platform has to efficiently handle massive *Concurrent iterative Graph Processing* (CGP) jobs. Although it has been extensively studied to optimize the execution of a single job, existing solutions face high ratio of data access cost to computation for the CGP jobs due to significant cache interference and memory wall, which incurs low throughput. In this work, we observed that there are strong spatial and temporal correlations among the data accesses issued by different CGP jobs because these concurrently running jobs usually need to repeatedly traverse the shared graph structure for the iterative processing of each vertex. Based on this observation, this paper proposes a correlations-aware execution model, together with a core-subgraph based scheduling algorithm, to enable these CGP jobs to efficiently share the graph structure data in cache/memory and their accesses by fully exploiting such correlations. It is able to achieve the efficient execution of the CGP jobs by effectively reducing their average ratio of data access cost to computation and therefore delivers a much higher throughput. In order to demonstrate the efficiency of the proposed approaches, a system called CGraph is developed and extensive experiments have been conducted. The experimental results show that CGraph improves the throughput of the CGP jobs by up to 2.31 times in comparison with the existing solutions.

1 Introduction

In the past decade, iterative graph analysis has become increasingly important in a large variety of domains [7, 26], which need to iteratively handle the graph round by round until convergence. Due to the increasing need of analyzing graph-structured data (e.g., social networks and web graphs), many iterative graph algorithms run as concurrent services on a common platform. These *Concurrent iterative Graph Processing* (CGP) jobs are usually executed on the same graph simultaneously so as to analyze it for various information. For example, facebook [2] uses Giraph [13] to handle a large number of CGP jobs (such as the variants of pagerank [21], SSSP [20], and k-means [16]) daily over the same graph (or its different snapshots) to provide various information for different products, respectively. Figure 1(a) gives the number of the CGP jobs traced over a large social network and shows that more than 20 CGP jobs may be submitted to the common platform to concurrently analyze the same graph in an iterative way at the peak time.



(a) Number of the CGP jobs

(b) Ratio of shared graph data

Figure 1: Information traced on a social network

Many systems are recently proposed to support large-scale graph analytics. They try to fully utilize high sequential memory bandwidth [17, 22, 23], improve data locality [11, 28, 31, 32], spare the redundant data accesses [6, 25], and reduce the memory consumption [29, 30], etc. Despite of these research efforts, there is a major challenge for the efficient execution of the CGP jobs. When a massive number of CGP jobs are running on the same underlying graph using the existing systems, each individual CGP job separately accesses the shared graph structure along different graph paths, resulting in repeatedly loading of the same data into the cache at different time by different jobs. They suffer from expensive data

[*]Corresponding Author: Xiaofei Liao

access overhead due to the factors such as serious cache interference and limited bandwidth. As the result of high ratio of data access cost to computation in graph algorithm, the current graph processing systems experience low throughput. This paper investigates whether and how we can improve the throughput of the CGP jobs.

In practice, the CGP jobs usually need to repeatedly traverse the shared graph and iteratively process each vertex for their own purpose. It suggests that there are a large number of intersections among the graph structure data being accessed by these jobs in each iteration, which we call the *spatial correlation* of data accesses. In addition, the partition of the shared graph structure may need to be accessed by multiple jobs within a short time interval, which we call the *temporal correlation* of data accesses. These two correlations indicate that there exist significant redundant data storing and accessing cost in the jobs, which leaves us good opportunities to reduce these unnecessary costs and improve the throughput.

Based on the observation, we propose a data-centric *Load-Trigger-Pushing* (denoted by LTP) model to improve the throughput of CGP jobs by fully exploiting the correlations of their data accesses. It decouples the graph structure data from the vertex state associated with each job. Within each iteration, the graph structure partitions shared by multiple CGP jobs are streamed into the cache and trigger the related jobs to concurrently process the data, followed by vertex state pushing for convergence. In this way, many accesses to the shared graph partitions can be amortized by multiple CGP jobs through handling them along a common order. The consumption of cache/memory is also reduced since a single copy of the graph structure data is used to serve multiple jobs at the same time. It indicates higher throughput thanks to much lower data access cost. To further improve the throughput, a core-subgraph based scheduling algorithm is designed to maximize cache utilization by judiciously arranging the loading order of the partitions.

We conducted the extensive experiments with our system CGraph and compare its performance with those of three cutting-edge graph processing systems, i.e., CLIP [6], Nxgraph [11], and Seraph [29, 30]. Experimental results show that CGraph improves the throughput of the CGP jobs by up to 3.29 times, 4.32 times, and 2.31 times over CLIP, Nxgraph, and Seraph, respectively.

The remainder of this paper is organized as follows. Section 2 discusses the the motivation of this work. Section 3 outlines our approach, followed by experimental evaluation in Section 4. Section 5 gives a survey of related work. Finally, we conclude this paper in Section 6.

2 Problem Presentation and Motivation

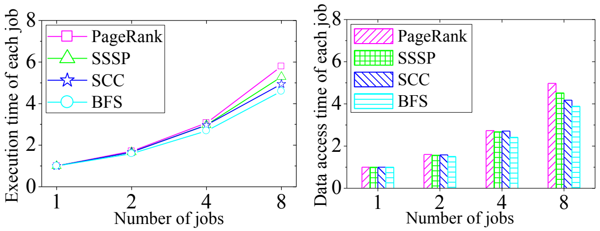
A common characteristic of an iterative graph processing job is that the operations are usually operated on two types of data: graph structure data and vertex state data. The graph structure data contains the edges between vertices and the information associated with each edge, whereas the vertex state data (e.g., ranking scores for PageRank [21], the distances from the source vertex for SSSP [20]) is computed by its tasks in a parallel way within each iteration and typically consumed in the next iteration. The graph structure data always occupies a majority of the memory, as compared with the vertex state data (i.e., job-specific data), and its proportion varies from 71% to 83% for different datasets [30]. As evaluated in Figure 1, the graph structure data is usually shared by multiple CGP jobs. However, in existing graph processing systems, these CGP jobs handle the shared graph in an individual manner along different graph paths, incurring low throughput for many redundant accesses to the shared graph and cache interference.

2.1 Data Access Problems of the CGP Jobs

In order to investigate the level of the inefficiency of the individual data accessing manner of the CGP jobs, we conducted the benchmark experiments to evaluate the execution time of different number of CGP jobs over Seraph [29, 30] on uk-union [3]. The hardware platform and benchmarks are the same as those described in Section 4.

From Figure 2(a), we made two observations. First, the concurrent way performs better than the sequential way of executing the jobs one by one. As observed in the experiments, it is because that the execution time of graph processing job is dominated by the data access cost and the CPU is always underutilized. Seraph is able to utilize the CPU in a better way by concurrently executing the jobs and also allowing them to share the in-memory graph structure data for less average data access cost. When there are eight jobs, the total execution time of the concurrent execution way is about 60% of the sequential way. Note that the total execution time of the concurrent way is the maximum value of these jobs' execution time, while it is the sum of those of all jobs for the sequential way. Second, the average execution time of each job, however, is significantly prolonged as the number of jobs increases. It is almost doubled when the number of jobs increases from four to eight, because of higher data access cost for each vertex processing.

Figure 2(b) shows the average data access time of the jobs over Seraph when the number of jobs increases. We can observe that the increment of the number of jobs leads to the significant rise of data access cost. It is because that the shared graph partitions are handled by the CGP jobs in an individual manner along different graph paths. As the number of the CGP jobs increases, more copies of the same data need to be created and loaded into the cache by the jobs at different time. Thus, more redundant data accesses are issued by the CGP jobs and



(a) Average execution time (b) Average data access time

Figure 2: Normalized performance of each CGP job over Seraph against that of the sequential way

it also incurs more serious cache interference due to the fact that more redundant data are stored into the cache for different jobs at different time. It eventually leads to low system throughput, since the data accessing cost typically represents a major proportion of the total execution time for an iterative graph algorithm.

Take Figure 3 as an example and assume that the cache can only store a partition for the CGP jobs. With the existing solutions, the SSSP job may firstly access partition 1 and then partition 2, whereas the PageRank job may firstly access partition 2 and then partition 1. Besides, the processing of each partition is various for different jobs, making their accesses more irregular. As a result, the partition 1 and partition 2 need to be repeatedly loaded into the cache. It leads to serious contention among the jobs for the data access channel, the cache and so on.

2.2 Correlations between the CGP Jobs

It is common that a set of CGP jobs involve in the analysis of the same graph. Figure 1(b) shows the ratios of a graph shared by different number of CGP jobs at various time sampled from the real trace. We discover that there are strong temporal and spatial correlations between the data accesses of the CGP jobs due to the repeated traverse of the graph shared by them. It indicates that many redundant accesses are issued by the CGP jobs and much cache space is also wasted to store several copies of the same graph structure data for the jobs at different time.

As described in Figure 1(b), the intersections of the set of graph partitions to be handled by different CGP jobs in each iteration are large (more than 75% of all active partitions on average). This is called the *spatial correlation*. However, the CGP jobs in existing systems access the shared graph partitions in different order individually, inducing much redundant overhead. Ideally, these CGP jobs should consolidate the accesses to the shared graph structure and store a single copy of the shared data in the cache to serve multiple CGP jobs at the same time.

In addition, some graph partitions may be accessed by multiple CGP jobs (may be more than 16 jobs) within a short time duration. This is called the *temporal correlation*. The traced results show that the number of CGP jobs to access each partition is skewed at any time. The current solutions, e.g., *Least Recently Used* (LRU) algorithm, may load the infrequently-used data into the cache when it is needed. It not only incurs the cost to load the data, but also swaps out the frequently-used data. A better solution should take into account the temporal correlations, e.g., the usage frequency of the graph partitions, when loading them into the cache.

These observations motivate us to develop a solution for efficient use of cache/memory and the data access channel to achieve a higher throughput by fully exploiting the spatial/temporal correlations discussed above.

3 Our Proposed Approach

Although we have identified the correlations between the CGP jobs, there are still several challenges that need to be tackled in order to exploit them. First, the shared vertices and edges may be individually handled by different jobs along different graph paths. Second, the CGP jobs have different properties (e.g., the rounds for convergence and the submission time), which reduce the chance of sharing the accesses to the graph structure data within a short time interval. For example, some graph structure partitions may be accessed by some jobs (e.g., PageRank) much more frequently than the others (e.g., BFS). Besides, the CGP jobs that share the same graph structure may be put into execution at different time. Third, it is a non-trivial task to design an efficient partition-loading order that can achieve a high cache utilization ratio.

Thus, we propose a data-centric *Loading-Trigger-Pushing* (LTP) model to fully exploit the spatial/temporal correlations between the CGP jobs, aiming to minimize the redundant accessing and storing cost of the shared graph structure data. In our LTP model, the shared graph is divided into a set of partitions. These partitions are loaded into the cache in sequence and in the same order for all jobs, where each partition is concurrently handled by the related CGP jobs. By such means, the accessing and storing of most graph structure partitions can be shared by multiple CGP jobs, thus significantly reducing the data access cost. When loading the graph partitions, a scheduling algorithm is further developed to specify the loading order of graph partitions (as well as the related job-specific data). The scheduler aims to maximize the cache utilization by fully exploiting the temporal correlations among the jobs' data accesses.

3.1 Data-centric LTP Execution Model

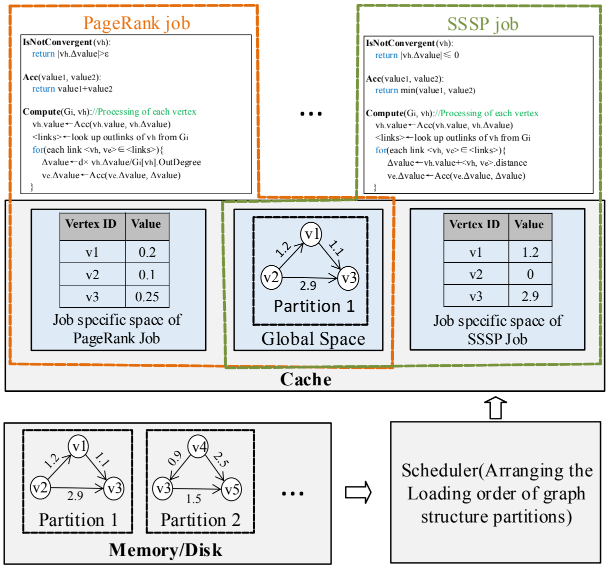
Assume that the data for an iterative graph algorithm is expressed as $D = (V, S, E, W)$, where V is the set of vertices, S is the set of states for the vertices, E is the set of edges, W is the set of weights associated with the edges. In our LTP model, the data of each job is decoupled as the graph structure data, i.e., $G = (V, E, W)$, and job-specific vertex states, i.e., S, where $G = \cup_i G^i$ is shared by different jobs and G^i is the i^{th} partition of the graph G. Each job has its own S, and repeatedly updates its S through its processing iterations until the calculated results converge. The processing of each iteration is divided into three stages: graph loading, parallel trigger, and pushing stage, which are formalized as follows.

Graph Loading. In each iteration, the shared graph structure partitions, e.g., G^i, are sequentially loaded for the CGP jobs along an order. It performs the following operation to load a graph partition: $L^i \leftarrow L(G^i, \cup_{j\in J} S^i_j)$, where $L(*)$ denotes an operator that loads the data specified in the parameter list "*" into the cache, J is the job set, S^i_j denotes the states of the vertices in G^i associated with the j^{th} job, and L^i is the data loaded into the cache. $S_j = \cup_i S^i_j$ is the set of vertex states related to the j^{th} job. In this way, it only needs to load a copy of each shared graph partition, e.g., G^i, for multiple CGP jobs and the partitions are also loaded for these jobs along a common order to provide opportunity to the sparing of redundant accesses by fully utilizing the correlations of these jobs.

Trigger and Parallel Execution. For each loaded graph partition G^i, the related CGP jobs, which are the jobs that need to process the vertices in the partition G^i and have not yet obtained the convergent results, are triggered to concurrently execute the following operator: $S^{i_{new}} \leftarrow \cup_{j\in J} T_j(G^i, S^i_j)$. The function $T_j(G^i, S^i_j)$ denotes the specific graph processing operations performed by the activated job j on the loaded data (i.e., G^i and S^i_j) towards its own objectives. Its outputs (denoted by $S^{i_{new}}_j$) are the new states related to the vertices in G^i and are associated with the j^{th} job. $S^{i_{new}}$=$\cup_{j\in J} S^{i_{new}}_j$ is the new vertex states that are related to the vertices in G^i for all CGP jobs. When the processing of G^i is finished for all related jobs, the next partition then can be loaded. By such means, it enables multiple jobs to regularly and concurrently process the set of shared graph partitions for their own goals along the same order and efficiently share the accesses to them for lower overhead.

State Pushing. If a job, e.g., j, has processed all its active partitions in an iteration, its new calculated results, i.e., S^{new}_j=$\cup_i S^{i_{new}}_j$, at this iteration are pushed for the state synchronization between the vertices of its different partitions (stored in its own job-specific space) for convergence. Then, the job starts a new iteration. Note that a CGP job will move to the next iteration once it has processed all active partitions in its current iteration and therefore different CGP jobs may be in different iterations of their graph processing. For example, BFS [10] may only need to handle a few active partitions in each iteration, while other algorithms, e.g., PageRank [21], may have to go through all partitions to complete an iteration.

Figure 3 gives an example to illustrate the LTP model. In this example, the graph in Figure 4(a) is divided into two partitions, which need to be handled by two jobs, i.e., a PageRank job and a SSSP job. The graph structure data is stored in the global space and is shared by these two CGP jobs, while the job-specific space is provided for each CGP job to store its own vertex states. It can load the two partitions along the order of partition 1 then 2. When the partition 1 and the related job-specific data are loaded into the cache, the related jobs (i.e., the PageRank job and the SSSP job) are triggered to concurrently handle it and update their own vertex states. When the two jobs have handled the partition 1, the partition 2 can be loaded for processing. When both the two partitions are handled by the jobs, the new iteration of each job begins.



PageRank job
SSSP job
...
Partition 1
Global Space
Job specific space of PageRank Job
Job specific space of SSSP Job
Cache
v1
v2
v3
1.2
1.1
2.9
Memory/Disk
Partition 1
Partition 2
v4
v5
0.9
2.5
1.5
Scheduler(Arranging the Loading order of graph structure partitions)

```
IsNotConvergent(vh):
  return |vh.Δvalue|>ε

Acc(value1, value2):
  return value1+value2

Compute(Gi, vh)://Processing of each vertex
  vh.value←Acc(vh.value, vh.Δvalue)
  <links>←look up outlinks of vh from Gi
  for(each link <vh, ve>∈<links>){
    Δvalue←d× vh.Δvalue/Gi[vh].OutDegree
    ve.Δvalue←Acc(ve.Δvalue, Δvalue)
  }
```

```
IsNotConvergent(vh):
  return |vh.Δvalue|⩽ 0

Acc(value1, value2):
  return min(value1, value2)

Compute(Gi, vh)://Processing of each vertex
  vh.value←Acc(vh.value, vh.Δvalue)
  <links>←look up outlinks of vh from Gi
  for(each link <vh, ve>∈<links>){
    Δvalue←vh.value+<vh, ve>.distance
    ve.Δvalue←Acc(ve.Δvalue, Δvalue)
  }
```

Vertex ID	Value
v1	0.2
v2	0.1
v3	0.25

Vertex ID	Value
v1	1.2
v2	0
v3	2.9

Figure 3: Illustration of our data-centric LTP model

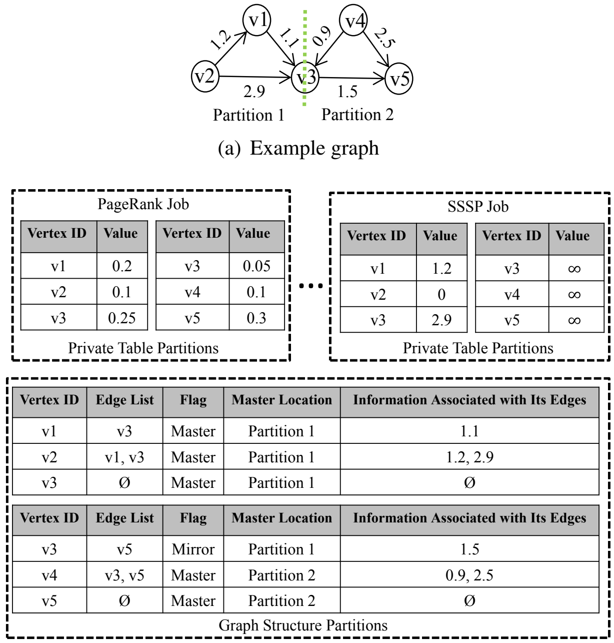
3.2 Correlations-aware Execution of Jobs

This section discusses how to efficiently implement our LTP model for the execution of multiple CGP jobs.

3.2.1 Graph Storage for Multiple CGP Jobs

We first show how to efficiently store the graph for the CGP jobs in our approach.

Data Structure of Graph Partition. For parallel processing, large-scale graph needs to be divided into partitions. However, the real-world graph usually has highly skewed power-law degree distributions [12], incurring imbalanced load among the partitions. Thus, our system also uses existing vertex-cut partitioning method [31], and evenly divides the edges of the graph into same-sized partitions in terms of the number of edges. Note that a vertex may have multiple replicas (e.g., v_3 in Figure 4(b)), where one of the replicas is nominated as the master vertex and the others are regarded as the mirror vertices. In this way, it not only gets balanced load for the partitions, but also does not incur communication cost when handling each partition. The communication only occurs when the replicas of the same vertex in different



(a) Example graph

PageRank Job

Vertex ID	Value
v1	0.2
v2	0.1
v3	0.25

Vertex ID	Value
v3	0.05
v4	0.1
v5	0.3

Private Table Partitions

SSSP Job

Vertex ID	Value
v1	1.2
v2	0
v3	2.9

Vertex ID	Value
v3	∞
v4	∞
v5	∞

Private Table Partitions

Vertex ID	Edge List	Flag	Master Location	Information Associated with Its Edges
v1	v3	Master	Partition 1	1.1
v2	v1, v3	Master	Partition 1	1.2, 2.9
v3	Ø	Master	Partition 1	Ø

Vertex ID	Edge List	Flag	Master Location	Information Associated with Its Edges
v3	v5	Mirror	Partition 1	1.5
v4	v3, v5	Master	Partition 2	0.9, 2.5
v5	Ø	Master	Partition 2	Ø

Graph Structure Partitions

(b) Details for the related tables

Figure 4: An example to show how to store data for multiple jobs, where the graph is divided into two partitions.

partitions synchronize their states. In order to effectively store the graph partitions for the CGP jobs, multiple key-value tables are established. In detail, a single global table is created to store the graph structure data for all CGP jobs. Multiple private tables are used to store the vertex states of the jobs, i.e., one table for each job.

Each global table entry represents a graph structure partition indexed by its key and with three other fields to describe corresponding information. The first two fields indicate the location of this graph structure partition and the number of its vertices, respectively. The third field stores the IDs of the jobs to process it at the next iteration (along with the locations of the related private tables associated with these jobs). The information of each graph structure partition is also stored in a key-value table and each of its data item indicates a vertex and contains four fields: vertex ID, edges assigned in this partition, flag, master location and the information associated with its edges, e.g., priority. Each private table entry represents a vertex state and has two fields, i.e., vertex ID and its state. Figure 4 gives an example to illustrate how to store the data for multiple jobs.

Suitable Size of Graph Partition. In order to efficiently use the parallelism of CPU and ensure good cache locality, the cache is expected to be just fully loaded when each core has data to handle. Therefore, the suitable size of each graph structure partition, i.e., P_g, is determined by the number of CPU cores, i.e., N, and the size of the cache, i.e., C. The value of P_g is expected to be the maximum integer such that $P_g + \frac{P_g}{s_g} \times s_p \times N + b \leq C$, where

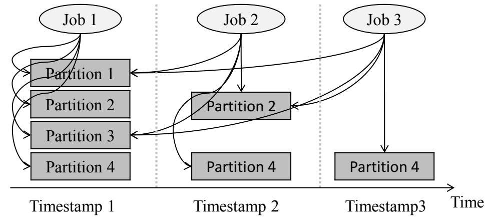


Figure 5: An example to illustrate how to store evolving graph structure for the CGP jobs submitted at different timestamps, where partitions 2 and 4 are changed at timestamp 2, and partition 4 is changed at timestamp 3.

s_g is the average size of each graph structure partition's item, s_p is the size of each private table's item, and b is the size of reserved buffer.

Details to Store Evolving Graph Structure. In practice, the graph structure may evolve with time. Thus, we also maintain a series of snapshots for it, where the graph updates, e.g., the adding/deleting of vertices and edges, are only visible to the jobs submitted later than the updates. In this way, different jobs are able to correctly handle the related snapshots of the graph, respectively. Because the changes of graph structure are usually very small at each time, the most part of these snapshots is the same. Thus, the series of snapshots can be stored in an incremental way for low overhead. For each snapshot, it creates a new global table and labels it with a timestamp, where this table only stores the new version of the partition with changes. The newly submitted job handles the graph partitions with the highest timestamp yet less than its arrival time. Figure 5 gives an example to illustrate it. Note that most graph structure partitions, e.g., the partitions 1 and 3, are usually shared by the jobs when they handle different snapshots, respectively.

3.2.2 Loading of Graph Partition

In practice, a partition is to be handled by a job in the next iteration only when its vertices are activated by the other ones of this job at the current iteration. Therefore, it is easy for each partition to identify the set of CGP jobs to process it within the next iteration through tracing the partitions activated within the current iteration. This profiled information, i.e., the temporal correlations of the jobs, is stored in the global table for each partition. The spatial correlations between the data accesses issued by the CGP jobs can be gotten by calculating the intersection of the set of graph partitions to be processed by different jobs. After that, it is able to load the shared graph partitions for exactly once along a common order to serve multiple CGP jobs within each iteration, amortizing the data access cost. Note that the correctness will not be affected by any loading order and the runtime loads the partitions in a round-robin way by default.

For each job, the states of most vertices may have converged at the early iterations, although some vertices

Algorithm 1 Details of each trigger

```
1: procedure TRIGGER($G^i$, $S^i_j$)
2:     for each $v_h \in S^i_j \wedge$ IsNotConvergent($v_h$) do
3:         Compute($G^i$, $v_h$)
4:         if $v_h$ is a mirror vertex then
5:             $D \leftarrow v_h.MasterLocation$
6:             $S^{new}_j$.Insert($v_h$, $i$, $D$, $v_h.\Delta value$)
7:         end if
8:     end for
9: end procedure
```

need hundreds of iterations for convergence. The loading and processing of the inactive vertices can be skipped for the related job for low overhead. In detail, when a graph structure partition G^i is loaded into the cache, it only loads the related job-specific private partitions, e.g., S^i_j, of the CGP jobs which need to process G^i. It does not load G^i when there is no job to handle G^i, i.e., the states of the vertices in G^i are inactive for all jobs. Specifically, when a graph structure partition is not used by any job at the next iteration, this graph structure partition is labeled as an inactive one so as to skip its loading. Similarly, it is relabeled as an active one when it needs to be processed by some jobs at the next iteration.

3.2.3 Parallel Processing of Graph Partition

After loading a graph partition G^i into the cache, it triggers the related CGP jobs (e.g., j) to concurrently handle their private vertex states (e.g., S^i_j) associated with this partition, respectively. Note that any newly submitted job only needs to register the partitions to be processed by it at its first iteration and waits to be triggered to handle them. It is possible that the number of CGP jobs is more than the number of CPU cores, i.e., N. Assume a partition is shared by $|J|$ number of jobs. When the value of $|J|$ is larger than N, these CGP jobs are assigned to be processed as different batches, where the shared graph structure partition is fixed in the cache and only the job-specific partitions are replaced. A graph structure partition is swapped out of the cache only when it has been processed by the related jobs within the current iteration. Otherwise, the unprocessed jobs need to load it again.

For the processing of each partition, the computation load of different CGP jobs is usually skewed, leading to low utilization ratio of hardware. In order to tackle this problem, it identifies the straggler, i.e., the job with the most number of unprocessed vertices in its private table for this partition. Note that the number of unprocessed vertices can be easily gotten, because the number of active vertices for each job in each partition is known as this partition is handled by the jobs at the previous iteration. Then, as described in Figure 6, it logically divides the unprocessed vertices in the private partition of the straggler into pieces and assigns them to the free cores to assist its processing.

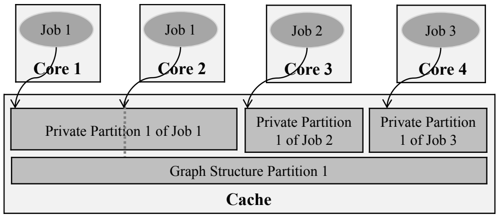

Job 1
Core 1
Job 1
Core 2
Job 2
Core 3
Job 3
Core 4
Private Partition 1 of Job 1
Private Partition 1 of Job 2
Private Partition 1 of Job 3
Graph Structure Partition 1
Cache

Figure 6: An example to illustrate how to get balanced load, where the core 1 and the core 2 are handling the partition 1 of the private table of the job 1 together.

The processing details for a job are given in Algorithm 1, where each job only computes the new state for its vertices in S^i_j according to their local neighbors recorded in the graph structure partition G^i (See Lines 2-8). Therefore, there is no cache miss when handling a partition, because no communication occurs between the vertices on different partitions. Obviously, the vertex with replicas on different partitions needs to synchronize their states. The mirror vertex needs to push its new state to its master vertex to get this vertex's final state at the current iteration. The new calculated state on the master vertex needs to be pushed to its mirrors. As a result, for such a vertex state synchronization, many partitions of private table are frequently loaded into the cache and incur high cache miss rate. In order to tackle this problem, for each mirror vertex, its new state is directly buffered in S^{new}_j (See Line 6), which will be implicitly sent to the master replica for batched vertex state synchronization at the data synchronization stage.

3.2.4 Data Synchronization

When there are multiple CGP jobs to synchronize vertex state, it is done for the jobs one by one to reduce resource contention, because there is no data sharing between the jobs. For efficient vertex state synchronization among replicas, as depicted in Algorithm 2, they are done together in batches at this stage for each job, aiming to avoid the frequent load of private table's partitions at runtime. The items buffered in the queue S^{new}_j (with the new states of the mirror vertices, e.g., v_h) are firstly sorted according to the IDs (e.g., $v_h.MasterLocation$, which is described in Figure 4(b)) of the partitions with the related master vertices (See Line 2), before pushing them.

By such means, it only needs to load fewer partitions of private table for the state updates of master vertices, since many updates become successive accesses to the same partition. Besides, when the successive updates for a master vertex are done (See Line 7), the final state of this vertex for the current iteration is gotten. Then, such a new value can be directly buffered for batched state updating of mirror vertices as well (See Lines 10-12). Note that, with the traditional solutions, it is impossible to

Algorithm 2 Details of data synchronization

```
1:  procedure PUSH($j$, $S_j^{new}$)
2:      SortD($S_j^{new}$) /*Sort the items recorded in $S_j^{new}$*/
3:      for <$v_h$, $i$, $MasterLocation$, $\Delta value$> $\in S_j^{new}$ do
4:          $D \leftarrow S_j^{new}[v_h].MasterLocation$
5:          $value \leftarrow S_j^{new}[v_h].\Delta value$
6:          $S_j^D[v_h].\Delta value \leftarrow Acc(S_j^D[v_h].\Delta value, value)$
7:          if Last update of $S_j^D[v_h].\Delta value$ is end then
8:              $val \leftarrow S_j^D[v_h].value$
9:              $S_j^D[v_h].value \leftarrow Acc(val, S_j^D[v_h].\Delta value)$
10:             for each $S_j^{new}[v_h].MasterLocation = D$ do
11:                 $S_j^{new}[v_h].\Delta value \leftarrow S_j^D[v_h].\Delta value$
12:             end for
13:         end if
14:     end for
15:     SortS($S_j^{new}$) /*Sort the items recorded in $S_j^{new}$*/
16:     for <$v_h$, $i$, $MasterLocation$, $\Delta value$> $\in S_j^{new}$ do
17:         $i \leftarrow S_j^{new}[v_h].i$
18:         $S_j^i[v_h].\Delta value \leftarrow S_j^{new}[v_h].\Delta value$
19:     end for
20: end procedure
```

know whether the final state is gotten for a master vertex until all updates are done. Then, the master vertex needs overhead to be reloaded for accessing, because it may have been swapped out of the cache. After that, it is done in a similar way to update mirror vertices' states according to the related master vertices' states (See Lines 15-19), where the items are sorted according to the IDs of the partitions with the mirror vertices (See Line 15).

3.3 Scheduling Based on Core-subgraph

With existing solutions, the partitions loaded into the cache may be underutilized. First, some vertices need more iterations to converge than the others for much higher degree. They make the partitions containing them repeatedly loaded into the cache and incur high overhead to load and store the early convergent vertices in the same partition. Second, the usage frequency of different partitions is also skewed and also evolving with time. In detail, the same partition of different jobs and different partitions in the same job all may need various iterations to converge. Besides, a graph partition is only visible to the jobs with the arrival time later than its timestamp. As a result, a loaded partition may need to be processed by very few (even one) jobs when the partition is arbitrarily loaded into the cache, inducing poor performance.

In order to maximize the utilization ratio of each partition loaded into the cache, we propose a scheduling algorithm based on core-subgraph partitioning. The key idea is to firstly put the core vertices (with degree higher than a given threshold) together and then make the loaded partition shared by as many jobs as possible on average via arranging the loading order of graph partitions. In detail, it firstly identifies a core subgraph, consisting of the core vertices and the edges on the paths between them, from the graph. Then it evenly divides the graph based on such a subgraph, where the edges of this subgraph are put together into several same-sized partitions and the remaining edges are divided into the other same-sized partitions. By such means, the frequently loading and processing of core vertices incur less cost to load the early convergent vertices in the same partition, sparing the consumption of bandwidth and the cache space.

After that, it gives each partition P a priority $Pri(P)$ and schedules the loading order of them based on the dynamically profiled priorities of them. The partition with the highest priority is firstly loaded into cache for the CGP jobs to handle, so as to improve the cache utilization ratio. The basic scheduling rules are as follows:

- First, a partition should be given the highest priority and be firstly loaded into the cache when it is needed by the most number of jobs for processing.
- Second, a partition should be set with a higher priority when it has a higher average vertex degree or a larger average vertex state changes, because more vertex states will be propagated to others through them. Then, most vertices need less iterations (also less consumption of the cache) to absorb other vertices' states for convergence.

The above rules are captured by such an equation:

$$Pri(P) = N(P) + \theta \cdot \overline{D}(P) \cdot C(P) \qquad (1)$$

where $N(P)$ is the number of jobs to process P and is used to capture the temporal correlations for the CGP jobs. $\overline{D}(P)$ is the average degree of the vertices in P, and $C(P)$ is the average state changes of the vertices in P for all its jobs at the previous iteration. The initial values of $N(P)$ and $C(P)$ and the value of $\overline{D}(P)$ are gotten at preprocessing time, while $N(P)$ and $C(P)$ are incrementally updated at the execution time. There, $0 \leq \theta < \frac{1}{\overline{D}_{max} \cdot C_{max}}$ is the scaling factor set by the runtime system at preprocessing time to ensure that a partition with the highest value of $N(P)$ is firstly processed, where $\overline{D}_{max}$ and C_{max} are the maximum values of any partition's $D(P)$ and $C(P)$, respectively. By such means, the partition loaded into the cache can serve as many jobs as possible, while the other partitions have more opportunity to be needed by more jobs after a time interval, further improving the throughput via reducing the average data access cost.

3.4 Implementation and Interfaces

The implementation details of CGraph are described in Algorithm 3. It repeatedly loads the unprocessed partitions, e.g., G^i, of the global table into the cache according to the scheduling algorithm (See Line 4). For each G^i,

Algorithm 3 Executor for CGraph

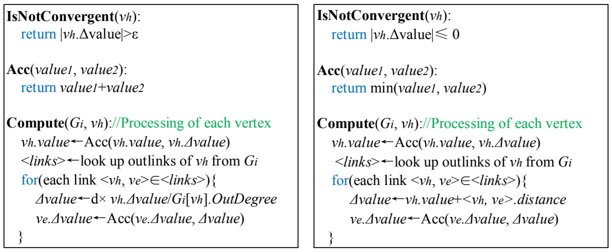
```
1:  procedure EXECUTOR($G$, $S_{Jobs}$)
2:      while the job set $S_{Jobs}$ is not empty do
3:          while $G$ has unprocessed $G^i$ for some jobs do
4:              $G^i \leftarrow$ LoadPartition($G$) /*Load $G^i$*/
5:              /*Get the set of jobs to handle $G^i$*/
                $J \leftarrow$ GetJobs($G^i$, $S_{Jobs}$)
6:              for each $j \in J$ do
7:                  /*Trigger the job $j$ to handle $G^i$*/
                    ParallelTrigger($j$, $G^i$, $S^i_j$)
8:              end for
9:              for each $j \in J$ and $S^{new}_j$ are gotten do
10:                 /*Vertex state synchronization for $j$*/
                    Push($j$, $S^{new}_j$)
11:                 if vertex states of $j$ are inactive then
12:                     /*Remove $j$ from the set $S_{Jobs}$*/
                        Remove($S_{Jobs}$, $j$)
13:                 end if
14:             end for
15:         end while
16:     end while
17: end procedure
```

the job-specific partitions, e.g., S^i_j, of the related CGP jobs are also loaded and these jobs are triggered to concurrently handle the loaded data (See Lines 5-8), where each job calculates the new states of its vertices according to the states of their local neighbors. When all active partitions of G have been handled for a job, e.g., j, at the current iteration (See Line 9), this job synchronizes the states of the vertices with several replicas distributed over different partitions (See Line 10). Then, its new iteration begins. Each job is repeatedly triggered until all its vertex states are inactive (See Lines 11-13). Note that it allows to add new jobs into S_{Jobs} at runtime.

For programming, a user only needs to instantiate three functions, i.e., IsNotConvergent(), Compute(), and Acc(), which are used in existing systems [23, 30, 31]. The first one indicates whether a vertex is convergent. Compute() is employed to update a vertex state and calculate the contributions of a vertex for the new states of its neighbors, and Acc() is utilized for a vertex to accumulate the contributions of its neighbors. Figure 7 gives two examples to show how to implement iterative graph algorithm on CGraph. Within each iteration, each vertex updates its state according to the accumulated contributions of its neighbors. After that, it calculates and sends its contributions to its neighbors for their state updates.

4 Experimental Evaluation

The hardware platform used in our experiments is a server containing 4-way 8-core 2.60 GHz Intel Xeon CPU E5-2670 and each CPU has 20 MB last-level cache, running a Linux operation system with kernel version 2.6.32. Its memory is 64 GB and the secondary storage for it is a disk with 1TB. It spawns a worker for each core to run benchmarks. The program is compiled with cmake version 2.6.4 and gcc version 4.7.2.



```
IsNotConvergent(vh):
   return |vh.Δvalue|>ε

Acc(value1, value2):
   return value1+value2

Compute(Gi, vh)://Processing of each vertex
   vh.value←Acc(vh.value, vh.Δvalue)
   <links>←look up outlinks of vh from Gi
   for(each link <vh, ve>∈<links>){
      Δvalue←d× vh.Δvalue/Gi[vh].OutDegree
      ve.Δvalue←Acc(ve.Δvalue, Δvalue)
  }
```

(a) PageRank

```
IsNotConvergent(vh):
   return |vh.Δvalue|⩽ 0

Acc(value1, value2):
   return min(value1, value2)

Compute(Gi, vh)://Processing of each vertex
   vh.value←Acc(vh.value, vh.Δvalue)
    <links>←look up outlinks of vh from Gi
   for(each link <vh, ve>∈<links>){
      Δvalue←vh.value+<vh, ve>.distance
      ve.Δvalue←Acc(ve.Δvalue, Δvalue)
  }
```

(b) SSSP

Figure 7: Instantiation of graph algorithms on CGraph

Table 1: Properties of data sets

Data sets	Vertices	Edges	Sizes
Twitter [3]	41.7 M	1.4 B	17.5 GB
Friendster [4]	65 M	1.8 B	22.7 GB
uk2007 [3]	105.9 M	3.7 B	46.2 GB
uk-union [3]	133.6 M	5.5 B	68.3 GB
hyperlink14 [5]	1.7 B	64.4 B	480.0 GB

In experiments, four popular iterative graph algorithms from web applications and data mining are employed as benchmarks: (1) PageRank [21]; (2) *single-source shortest path* (SSSP) [20]; (3) *strongly connected component* (SCC) [14]; (4) *breadth-first search* (BFS) [10]. The datasets used for these graph algorithms are real-world graphs existing on the websites [3, 4, 5] as described in Table 1. The performance of CGraph is compared with three cutting-edge graph processing solutions, i.e., CLIP [6], Nxgraph [11], and Seraph [29, 30], implemented by us on GridGraph [32]. Seraph is the state-of-the-art system optimized to support the efficient execution of multiple CGP jobs. Note that the jobs (e.g., PageRank, SSSP, SCC, and BFS) in the experiments are submitted to each system simultaneously.

4.1 Performance of Scheduling Strategy

First, we discuss the contributions of our scheduling algorithm on the performance of CGraph. In order to get this goal, PageRank, SSSP, SCC and BFS are executed as four CGP jobs to evaluate the total execution time of them over CGraph (with our scheduler described in Section 3.3) and CGraph-without (without our scheduler), respectively. Note that the graph partitions in CGraph-without are loaded in a round-robin way. As shown in Figure 8, the execution time of CGraph-without is more than that of CGraph under any circumstances. The execution time of CGraph is even only 60.5% of CGraph-without over hyperlink14. It is because that the scheduling scheme is able to maximize the utilization ratio of each partition in the cache via firstly loading the most important partition for the jobs.

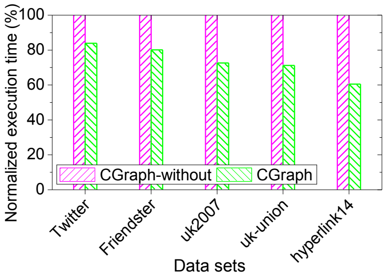


Figure 8: Execution time for the four jobs without/with our scheduler

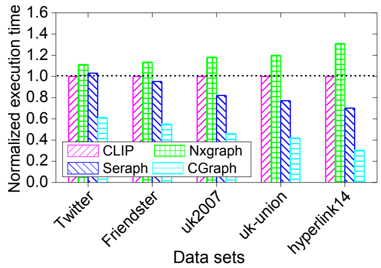


Figure 9: Total execution time for the four jobs with different solutions

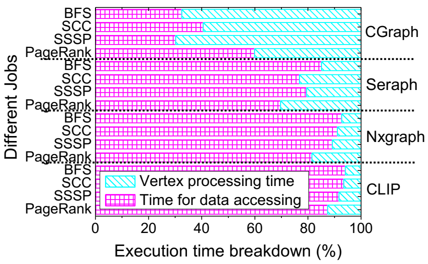


Figure 10: Execution time breakdown of different jobs on hyperlink14

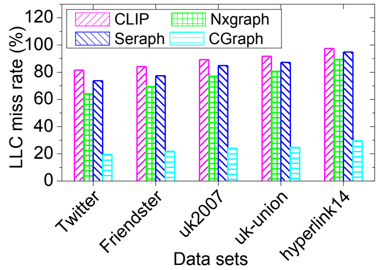


Figure 11: Last-level cache miss rate for the four jobs

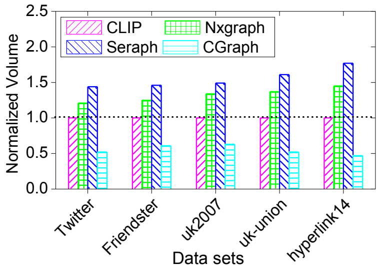


Figure 12: Volume of data swapped into the cache for the four jobs

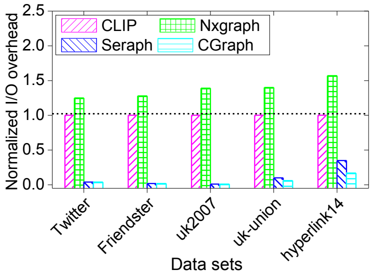


Figure 13: I/O overhead for the four jobs with different solutions

4.2 Overall Performance Comparison

To compare CLIP, Nxgraph, Seraph, and CGraph, we simultaneously submit PageRank, SSSP, SCC, and BFS as four jobs to each of these systems. Figure 9 shows the total execution time of the four jobs over different systems. We find that the four jobs over CGraph are able to converge with less time, which indicates higher throughput than the other systems. For example, over hyperlink14, CGraph can improve the throughput by 2.31 times, 3.29 times, and 4.32 times in comparison with Seraph, CLIP, and Nxgraph, respectively. We identify that the highest throughput of CGraph mainly comes from much lower average data access cost to computation ratio than them.

Figure 10 depicts the execution time breakdown of different jobs evaluated on hyperlink14 with different solutions. We can observe that the pure vertex processing time of the job over CGraph occupies the most ratio of its total execution time, while the ratio is very low in CLIP, Nxgraph, and Seraph. There are two reasons leading to lower average data access cost to computation ratio for CGraph than the other solutions. First, through efficiently exploiting the data access correlations between the CGP jobs, CGraph needs to store less data into the cache, getting a lower cache miss rate. Second, CGraph needs to access less volume of data due to efficient share of data accesses for the jobs, which means less consumption of bandwidth for main memory and the disk.

In order to demonstrate it, we firstly evaluate the last-level cache miss rates of CLIP, Nxgraph, Seraph, and CGraph using Cachegrind [1]. The miss rates of the above four jobs over them are given in Figure 11. As described, the cache miss rate of CLIP is larger than that of Nxgraph, because CLIP tries to trade off locality for the reduction of the total amount of data accesses while Nxgraph uses destination-sorted sub-shard structure to store a graph for better locality. However, the cache miss rate of Nxgraph is still more than that of CGraph. For example, the cache miss rate of Nxgraph is 89.5% for hyperlink14, while the rate is only 29.6% for CGraph. It is mainly because that a single copy of graph structure data in the cache is able to serve multiple jobs of CGraph.

Next, we evaluate the total volume of data swapped into the cache for the above four jobs over different systems. The normalized results of them against CLIP are depicted in Figure 12. We can find that CLIP needs to swap much smaller volume of data into the cache than Nxgraph and Seraph, because it is able to reduce the number of iterations for convergence via reentry of loaded data and beyond-neighborhood accesses. Note that the method employed by CLIP can also be used in Seraph as well as CGraph, rather than Nxgraph.

Besides, from Figure 12, we can observe that the volume of CGraph is much less than those of the other solutions. For example, the value of CGraph is only 47.1% of CLIP over hyperlink14, because CGraph does not need to load and to store the shared graph structure data for each job, separately. However, CLIP suffers from many redundant data accesses due to ignoring the data access correlations between the CGP jobs. Although Seraph can spare some data accesses from the disk to the main memory via sharing in-memory data, each job loads data into the cache in an individual way, incurring high data access overhead as well. It also means that Seraph is only suitable to out-of-core computation.

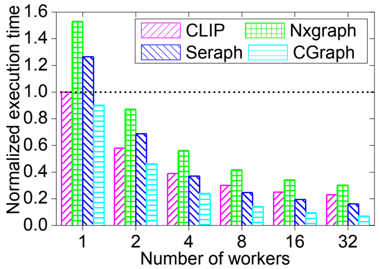


Figure 14: Scalability of the four jobs on hyperlink14

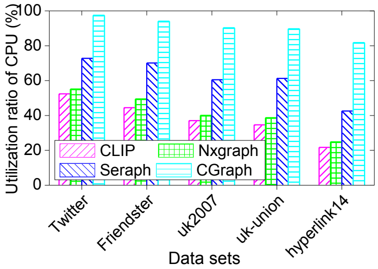


Figure 15: Utilization ratio of CPU for the four jobs

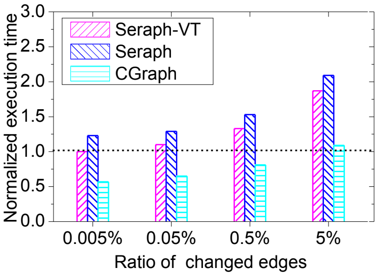


Figure 16: Execution time of the eight jobs on hyperlink14 with changes

Finally, the I/O overhead of the above four jobs is also evaluated over different systems. As shown in Figure 13, the jobs on the first three graphs almost not incur I/O overhead for both CGraph and Seraph, because they only needs to store one copy of the graph structure data in the main memory and these graphs can be totally loaded. When the graph size is larger than the memory size, CGraph needs less I/O overhead than Seraph through consolidating data accesses for the jobs. It also indicates a better performance of CGraph for out-of-core computation, because the data access time dominates the total execution time under such circumstances.

4.3 Scalability of CGraph

The scalability of CGraph is evaluated via executing the above four jobs on hyperlink14 and increasing the number of workers. Figure 14 gives the results relative to that of CLIP with only one worker. We can observe that CGraph has much better scalability than the other ones. The best scalability of CGraph mainly comes from efficient share of data accesses, while the other systems suffer from limited bandwidth for main memory and magnetic disk. Meanwhile, such a limited bandwidth also induces low utilization ratio of CPU for them.

In Figure 15, we evaluate the average utilization ratio of CPU for the vertex processing of the four jobs over different systems. As observed, existing systems suffer from low CPU utilization ratio, because the long data access time leads to the waste of CPU for them. Besides, from Figure 14 and Figure 15, we can find that the CPU cores of CGraph are almost fully utilized due to balanced load and low data access cost to computation ratio. It indicates that the limited computation ability of the CPU cores becomes the bottleneck of CGraph. GPGPU may be a suitable accelerator to help CGraph to process the CGP jobs for its powerful computing ability.

4.4 Performance on Graph with Changes

In real-world applications, several snapshots may be created for the graph with changes, and multiple CGP jobs are generated to handle them, respectively. In this part, we evaluate CGraph for the graph with structure changes. In the following experiments, we repeatedly generate the CGP jobs in the order of PageRank, SSSP, SCC, and BFS until a given number of jobs are created, where the CGP jobs are executed over a series of snapshots, respectively. Note that Seraph-VT is the version of Seraph incorporating multi-version switching approach [15].

First, we evaluate the total execution time of eight CGP jobs over different systems for hyperlink14 with the graph change ratio ranging from 0.005% to 5%. In detail, the change on the successive two snapshots ranges from 0.005% to 5%. Figure 16 gives the results relative to the execution time of Seraph-VT when the ratio of changed edges is 0.005%. We can observe that CGraph always gets the best performance under different graph change ratios. It is because CGraph still gets a low average data access cost to computation ratio, although the snapshots have differences in graph structure. Besides, we can also find that CGraph needs longer execution time to handle the graph when the graph change ratio is larger, because of less data access correlations between the jobs.

In the following experiments, we take a series of snapshots of hyperlink14 as datasets, where the change ratio between any successive two snapshots is 5% and each job handles a snapshot. Figure 17 depicts the execution time breakdown of the jobs over different systems on hyperlink14 when the the number of jobs increases. We can find that the jobs over CGraph have a lower average data access cost to computation ratio with the increase of the number of jobs, because there are more jobs to amortize the data access cost. However, for Seraph-VT and Seraph, the condition with more jobs leads to much more volume of data loaded into the cache and makes them suffer from serious cache interference and limited bandwidth. Thus, CGraph performs much better than Seraph-VT and Seraph when the number of jobs is larger.

The last-level cache miss rate is also evaluated for them on hyperlink14. As depicted in Figure 18, the cache miss rate of CGraph is significantly reduced when the number of jobs is increased, because the data in the cache can be repeatedly used by different CGP jobs. For example, in CGraph, the cache miss rate of the condition with eight jobs is even only 32.8% of the condition with one job. However, in the other solutions, the cache miss rate is significantly increased when the number of jobs is more, because of serious cache interference.

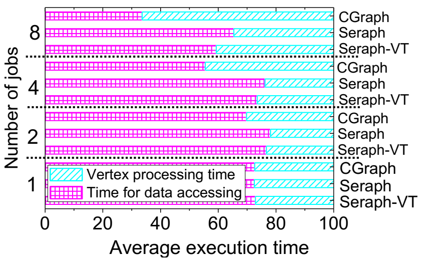


Figure 17: Execution time breakdown of different solutions on hyperlink14

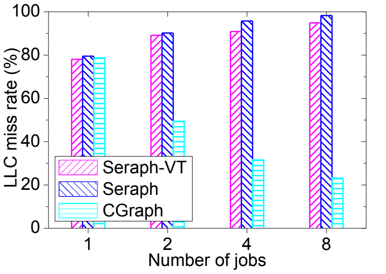


Figure 18: Last-level cache miss rate of different solutions on hyperlink14

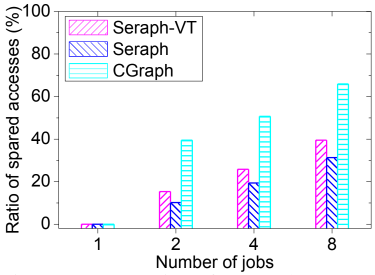


Figure 19: Ratio of spared accessed data on hyperlink14

Figure 19 gives the ratio of the total accessed data (including the data from the disk to the main memory and the main memory to the cache) spared by different solutions on hyperlink14 in comparison with the way sequentially executing the jobs over Seraph. As expected, the number of data accesses spared by CGraph is much more than the other solutions. For example, when the number of jobs is eight, the ratio is even up to 65.9% for CGraph, while the ratios of Seraph-VT and Seraph are only 39.5% and 31.3%, respectively. Besides, as observed, CGraph spares much more data accesses when the number of jobs increases, due to more opportunity to share data accesses between different jobs.

5 Related Work

With the explosion of graph scale, many systems [18, 19, 27] have focused on achieving high efficiency for iterative graph analysis. However, most of them focus on supporting single graph processing job. They improve the efficiency either by fully utilizing the sequential usage of memory bandwidth, or by achieving a better data locality and less redundant data accesses, which consequently reduces the volume of the accessed data.

GraphChi [17] achieves sequential storage access by employing parallel sliding windows. X-Stream [23] and Chaos [22] improve GraphChi by using streaming partitions for better sequential access of out-of-core data. Xie etc. [28] propose a novel block-oriented computation model, in which computation is iterated locally over blocks of highly connected nodes, which improves the amount of computation per cache miss. PathGraph [31] models a large graph using a collection of tree-based partitions for better locality. GridGraph [32] proposes 2-Level hierarchical partitioning scheme to improve the locality and reduce the amount of I/Os. NXgraph [11] uses destination-sorted sub-shard structure to store graph for better locality and adaptively chooses the fastest strategy to fully utilize memory and reduce data transfer. Instead of targeting a better locality, CLIP [6] proposes to reduce the total data access cost through the reentry of the loaded data and beyond-neighborhood accesses.

Although these systems can support efficient execution of a single iterative graph processing job, multiple separate copies of the same graph need to be created in the main memory by them for the CGP jobs. Following on this direction, Seraph [29, 30] is designed to allow multiple jobs to correctly share one copy of the in-memory graph structure data. However, in Seraph, the accesses to the same graph partitions are performed separately by the jobs along different graph paths, incurring redundant accesses and wasting the cache as well. Note that graph databases [8] are recently proposed to support concurrent queries over a graph. For example, TAO [9] provides a simple data model and APIs to store and query graph data. Wukong [24] uses a RDMA-based approach to provide low-latency concurrent queries over large graph-based RDF datasets. However, these graph database solutions can not efficiently support the execution of the CGP jobs because they are dedicated to graph queries which usually only touch different small subsets of a graph for exactly once, instead of iteratively processing the entire graph for many rounds.

6 Conclusion

This paper discovers that many redundant data accesses exist in the CGP jobs for their strong temporal and spatial correlations. A novel data-centric LTP model and an efficient scheduling algorithm is then proposed to exploit our observed data access correlations in these jobs and allows multiple CGP jobs to efficiently amortize the data access cost for higher throughput. Experimental results show that our approach significantly improves the throughput for the CGP jobs against the state-of-the-art solutions. This work mainly focuses on static graph processing. In the future, we will research how to further optimize our approach for evolving graph analysis and also extend it to distributed platform and also heterogeneous platform consisting of GPUs so as to get higher throughput for the CGP jobs.

Acknowledgments

This paper is supported by National Key Research and Development Program of China under grant No. 2018YFB1003500, National Natural Science Foundation of China under grant No. 61702202 and 61628204, China Postdoctoral Science Foundation Funded Project under grant No. 2017M610477 and 2017T100555.

References

[1] Cachegrind. http://www.valgrind.org/, 2017.

[2] facebook. http://www.facebook.com/, 2017.

[3] Law. http://law.di.unimi.it/datasets.php, 2017.

[4] Snap. http://snap.stanford.edu/data/index.html, 2017.

[5] Wdc. http://webdatacommons.org/hyperlinkgraph/, 2017.

[6] AI, Z., ZHANG, M., WU, Y., QIAN, X., CHEN, K., AND ZHENG, W. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *Proceedings of the 2017 USENIX Annual Technical Conference* (2017), pp. 125–137.

[7] BALUJA, S., SETH, R., SIVAKUMAR, D., JING, Y., YAGNIK, J., KUMAR, S., RAVICHANDRAN, D., AND ALY, M. Video suggestion and discovery for youtube: Taking random walks through the view graph. In *Proceedings of the 17th International Conference on World Wide Web* (2008), pp. 895–904.

[8] BORNEA, M. A., DOLBY, J., KEMENTSIETSIDIS, A., SRINIVAS, K., DANTRESSANGLE, P., UDREA, O., AND BHATTACHARJEE, B. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), pp. 121–132.

[9] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., AND LI, H. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), pp. 49–60.

[10] BULUÇ, A., AND MADDURI, K. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), pp. 1–12.

[11] CHI, Y., DAI, G., WANG, Y., SUN, G., LI, G., AND YANG, H. Nxgraph: An efficient graph processing system on a single machine. In *Proceedings of the 2016 IEEE International Conference on Data Engineering* (2016), pp. 409–420.

[12] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th Usenix Conference on Operating Systems Design and Implementation* (2012), pp. 17–30.

[13] HAN, M., AND DAUDJEE, K. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment 8*, 9 (2015), 950–961.

[14] HONG, S., RODIA, N. C., AND OLUKOTUN, K. On fast parallel detection of strongly connected components (scc) in small-world graphs. In *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis* (2013), pp. 1–11.

[15] JU, X., DAN, W., JAMJOOM, H., AND KANG, G. S. Version traveler: Fast and memory-efficient version switching in graph processing systems. In *Proceedings of the 2016 USENIX Annual Technical Conference* (2016), pp. 523–536.

[16] KUMAR, R., KUMAR, R., LU, K., VASSILVITSKII, S., AND VASSILVITSKII, S. Local search methods for k-means with outliers. *Proceedings of the VLDB Endowment 10*, 7 (2017), 757–768.

[17] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (2012), pp. 31–46.

[18] MAASS, S., MIN, C., KASHYAP, S., KANG, W., KUMAR, M., AND KIM, T. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the 12th European Conference on Computer Systems* (2017), pp. 527–543.

[19] MALICEVIC, J., LEPERS, B. J. E., AND ZWAENEPOEL, W. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *Proceedings of the 2017 USENIX Annual Technical Conference* (2017), pp. 631–643.

[20] MEYER, U. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Proceedings of the 12th annual ACM-SIAM Symposium on Discrete Algorithms* (2001), pp. 797–806.

[21] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project, 1998.

[22] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 410–424.

[23] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), pp. 472–488.

[24] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), pp. 317–332.

[25] VORA, K., XU, G., AND GUPTA, R. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *Proceedings of the 2016 USENIX Annual Technical Conference* (2016), pp. 507–522.

[26] WANG, K., HUSSAIN, A., ZUO, Z., XU, G., AND AMIRI SANI, A. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), pp. 389–404.

[27] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. Gram: Scaling graph computation to the trillions. In *Proceedings of the 6th ACM Symposium on Cloud Computing* (2015), pp. 408–421.

[28] XIE, W., WANG, G., BINDEL, D., DEMERS, A., AND GEHRKE, J. Fast iterative graph computation with block updates. *Proceedings of the VLDB Endowment 6*, 14 (2013), 2014–2025.

[29] XUE, J., YANG, Z., HOU, S., AND DAI, Y. Processing concurrent graph analytics with decoupled computation model. *IEEE Transactions on Computers 66*, 5 (2017), 876–890.

[30] XUE, J., YANG, Z., QU, Z., HOU, S., AND DAI, Y. Seraph: An efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing* (2014), pp. 227–238.

[31] YUAN, P., ZHANG, W., XIE, C., JIN, H., LIU, L., AND LEE, K. Fast iterative graph computation: A path centric approach. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), pp. 401–412.

[32] ZHU, X., HAN, W., AND CHEN, W. Gridgraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference* (2015), pp. 375–386.

Don't share, Don't lock: Large-scale Software Connection Tracking with Krononat

Fabien André
Technicolor

Stéphane Gouache
Technicolor

Nicolas Le Scouarnec
Technicolor

Antoine Monsifrot
Technicolor

Abstract

To simplify software updates and provide new services, ISPs are interested in migrating network functions implemented in residential gateways (such as DSL or Cable modems) to the cloud. Two key functions of residential gateways are Network Address Translation (NAT) and stateful firewalling, which both rely on connection tracking. To date, these functions cannot be efficiently implemented in the cloud: current OSes connection tracking is unable to meet the scale and reliability needs of ISPs, while hardware appliances are often too expensive. In this paper, we present Krononat, a distributed software NAT that runs on a cluster of commodity servers, providing a cost-efficient solution with an excellent reliability. To achieve this, Krononat relies on 3 key ideas: (i) sharding the connection tracking state across multiple servers, down to the core level; (ii) steering traffic exploiting the features of entry-level switches; and (iii) avoiding all locks and data sharing on the data path. Krononat supports a rate of 77 million packets per second on only 12 cores, tracking up to 60M connections. It is immune to single node failures, and supports elastic workloads by a fast reconfiguration mechanism ($<$ 500ms).

1 Introduction

Over the last years, network and datacenter operators have started *virtualizing network functions* such as routers, firewalls or load balancers. Network Function Virtualization (NFV) consists in replacing network functions implemented in hardware appliances by software implementations deployed on commodity servers. Thus, major companies such as Google or cloud providers such as Microsoft Azure rely on software implementations for their load balancing needs [9, 29]. In these cases, software implementations bring a number of benefits: (i) a better scalability than hardware devices, thanks to the use of a scale-out model, (ii) better redundancy properties and (iii) a higher flexibility, as new software can be easily deployed while hardware is hard to change.

Because of its benefits, Internet Service Providers (ISPs) are also embracing NFV. ISPs have expressed a growing interest in moving network functions implemented in residential gateways (also known as DSL, cable or fibre modems) to commodity servers in the core network. In addition to the physical layer, residential gateways usually implement tunneling, stateful firewalling, and Network Address Translation (NAT). While the physical layer and tunneling have to be implemented in the gateway, there is an opportunity for moving the firewall and NAT functions to commodity servers in the core network. For an ISP, this brings two main benefits: (i) simplifying updates by running software on a few servers rather than millions of gateways spread across a country, (ii) exposing the user's local network, creating opportunities for new services or troubleshooting.

If NFV brings many benefits to ISPs, it also comes with two critical challenges: cost efficiency and reliability. In current OSes, the performance of connection tracking and firewalls (such as netfilter in Linux) is such that deploying them at the scale of an ISP would require a prohibitive amount of servers. Moreover, they only provide limited options for fault tolerance, making them unable to meet the reliability requirements of large ISPs. In this paper, we tackle the problem of connection tracking at the scale of an ISP. We introduce Krononat, a distributed high-performance software stateful firewall and NAT that is able to meet the requirements of an ISP. This paper makes the following contributions:

- We highlight the features of modern CPUs and commodity server hardware architectures that enable the design of a resource-efficient connection-tracking system. We propose a hardware platform able to serve millions of users.
- We propose three software design principles that enable the construction of efficient network functions: (i) sharding the connection-tracking

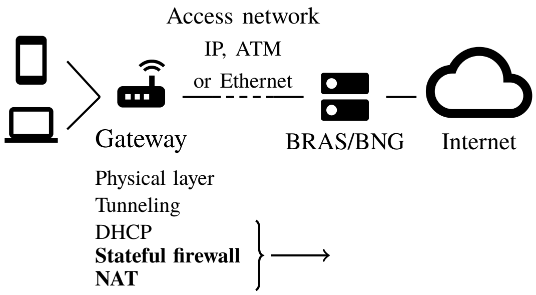


Figure 1: Simplified view of an ISP access network

	Without NFV	With NFV
Tunnel source IP	172.17.128.1	172.17.128.1
Tunnel destination IP	172.17.128.2	172.17.128.2
Source IP	**198.51.100.1**	**192.168.0.3**
Destination IP	203.0.113.1	203.0.113.1
	Payload	Payload

Figure 2: Tunneled packets on the access network

state down to the core level, (ii) using entry level switches to steer traffic to specific cores in multi-servers systems and (iii) avoiding all locks on the data path. We show how these principles are implemented in Kronanat's software architecture.

- We show that Krononat can manage 60M flows, corresponding to an aggregated total throughput of 70 Mpps (equivalent to 155 Gbps Simple IMIX traffic), on just 12 cores (spread on 4 servers).

2 Background

In this section, we quickly review what NAT and stateful firewalls are, and explain their relationship with connection tracking. We then give a simplified overview of an ISP network to show where our software, Krononat, fits.

Connection Tracking is an essential building block for numerous network functions. In this paper, we focus on two of them (i) NAT and (ii) stateful firewall.

We are interested in port-restricted cone NAT [35], which is commonly implemented in residential gateways. NAT is used to address the scarcity of IPv4 addresses by making several devices (e.g., laptops, smartphones etc.) with local IP addresses (typically in the 192.168.0.0/24 range) appear as a single IP address on the internet, the public address of the gateway. Once a connection between (local address; local port) and (external address; external port) has been established, the NAT: (i) sends every packet from (local address; local port) through (public address; public port), and (ii) sends packets sent to (public address; public port) to (local address; local port) if and only if they go to/originate from (external address; external port). NAT thus requires tracking connections by storing entries in a table.

A stateful firewall allows incoming traffic that belongs to connections established by a local host and rejects all other incoming traffic. More formally, a stateful firewall allows an incoming packet for an external socket (external address; external port) if and only if a connection between a local socket (local address; local port) and this external socket (external address; external port) was previously established. Just like NAT, stateful firewalls require to track established connections in a connection-tracking table. Stateful firewalls are an essential security measure useful for both IPv4 and IPv6 Internet access.

Residential Access Networks We depict a simplified architecture of a residential access network on Figure 1.

The first component of an ISP network is the user's residential gateway. It is located at the user side and is known as the Customer Premises Equipment (CPE). Local devices (smartphones, laptops etc.) are connected to the gateway either through wired Ethernet or Wi-Fi. The gateway also connects to the ISP access network (xDSL, Cable or Fiber). In addition to implementing the physical layer, the residential gateway performs a number of network functions. First, the gateway attributes IP addresses to local devices through DHCP. The gateway also restrict incoming traffic to connections established by local devices (stateful firewall). Lastly, the gateway performs Network Address Translation (NAT), so that local devices appear as a single IP address on Internet (Fig. 1).

The access network carries customer traffic from the DSLAM (DSL Access Multiplexer) or OLT (Optical Line Termination) to the Broadband Remote Access Server (BRAS). Customer traffic is tunneled using PPP, L2TP or GRE and transported over IP, ATM or Ethernet. Krononat uses GRE but can be easily adapted to other tunneling protocols. For GRE, the original IP packet (inner, grayed on Figure 2) is encapsulated into another IP packet (outer, white) for routing on the access network.

The Broadband Remote Access Server (BRAS) also named Broadband Network Gateway (BNG) collects customer traffic in a central location. The BRAS decapsulates tunneled packets and forwards them to Internet routers. With NFV, ISPs are moving network functions from the gateway (DHCP, Firewall, NAT) to the BRAS. Low-traffic functions (e.g., DHCP which handles a few packets per hour) are easy to move to the BRAS. By contrast, for firewall and NAT, *every* packet must be checked against the connection-tracking table, requiring the infrastructure to handle millions of packets per second.

ISP Contraints are unique and challenge existing NAT solutions. We review the most decisive ones.
Scale: ISPs operate at a very large scale: they have millions of simultaneously connected customers, which

CPU	2x Intel Xeon E5-2698v4 2x 20 cores, 2.2 Ghz 2x 40 PCIe v3 lanes
Memory	128 GiB (>100 GB/s bandwidth)
NICs	10x Intel XL710 40Gbps **400Gbps Total Throughput**
Price	$30000

Table 1: Commodity server for high-performance NAT

translates into dozens of millions of connections. In addition, at peak hours, they forward traffic in excess of 100Gbps. Thus, any solution should be computationally and cost efficient, i.e., the cost per user should be low.
Reliability: NAT service is crucial for Internet connectivity, as a disruption of NAT service translates into a loss of Internet connectivity for users. An ISP NAT solution must therefore be highly reliable, and should continue working in the event of a node failure.
Elasticity: ISPs operate in a limited geographical area and have a load with strong diurnal patterns [21, 12, 32, 36]. Thus, there is an opportunity to reduce the operating cost by dynamically adapting the number of servers.

Software NAT solutions OSes offer NAT functionality, usually implemented in kernel-mode (e.g., netfilter on Linux). However, a single node is not able to handle the load generated by all users of an ISP, and these implementations do not offer easy ways to aggregate multiple servers (i.e., distribute load accross servers). Projects such as Residential Cord [2] have been started to allow the use of multiple servers but remain impaired by the low computational efficiency of OSes NAT implementations, which translates into a high consumption of computing resources and high costs. In Residential Cord [2], their Linux-based vSG (virtual Service Gateway) hits a memory limit at 2000-4000 users per server. In [25], Linux NAT achieves 200 kpps (kilo packets per second) using a single core and 1 Mpps with 8 cores while BSDs achieve 220 kpps using 1 core and 500 kpps using 8 cores of an Intel Atom C2758 processor.

Appliances	Vendor A	Vendor B	COTS server
Max. Throughput	130 Gbps	140 Gbps	400 Gbps
Max. Connections	76M	180M	1000M
List Price	$65000	$200000	$30000
Price / Gbps	**$500**	**$1400**	**$75**

Table 2: NAT Hardware Solutions

Hardware NAT solutions Major vendors offer NAT solutions that can accommodate the traffic generated by a high number of users. However, these solutions tend to have a high cost (see Table 2). Also, they lack elasticity: the addition of a device requires manual configuration. They offer limited reliability options (often limited to 1+1 redundancy). Lastly, these solutions rely on tightly coupled specialized processors and specialized software. They therefore do not offer the flexibility of a full software solution, slowing down the addition of new features and preventing independent sourcing of hardware.

3 Designing a Software NAT

In this paper, we present Krononat, a multi-user stateful firewall and NAT service. Krononat is distributed on multiple servers so that it can handle the load generated by millions of users. Krononat groups users into *shards* that are dynamically mapped on servers. Krononat ensures that a server failure does not cause service disruption by replicating the state for a shard on two servers (a slave and a master). Krononat sits at the BRAS level and thus receives IP/GRE tunneled traffic and forwards NAT-ed packets to the Internet, and handles reverse traffic.

Our implementation builds on DPDK [1] and supports IPv4; yet our design generalizes to IPv6. We show that a careful software design and adequate hardware choice, allows achieving a high performance, and thus low-cost operations, without jeopardizing fault-tolerance.

3.1 Hardware Architecture

Current commodity servers and switches offer an opportunity for building NAT solutions that are competitive in performance with specialized solutions. Generally, specialized network appliances offer better performance than general-purpose servers through the use of content addressable memories (e.g., TCAM), that are notably used in routers and switches. However, maintaining connection-tracking tables requires much more memory than maintaining routing or switching tables: several gigabytes for connection tracking tables compared to tens of megabytes for routing tables. As TCAM are strongly limited in size (< 100 Mb), network appliances such as those of Table 2 must store connection-tracking tables in DRAM. Thus, for connection tracking, network appliances do not have a decisive advantage over commodity servers, which also use DRAM. Moreover, modern CPUs compensate memory latency by caches and out-of-order execution.

Thus, to minimize the cost, we rely on commodity servers equipped with a large number of Network Interface Cards (NICs). Current dual Intel Xeon servers offer 40 PCIe lanes, enough to handle 10 40-Gbps NICs, for a total throughput of 400 Gbps. For a typical Internet workload (Simple IMIX), this corresponds to 180 Mpps. An optimal NAT solution must therefore process at least 4.5 Mpps *per core*, so that one server (Table 1) can forward 400 Gbps, thus saturating its NICs.

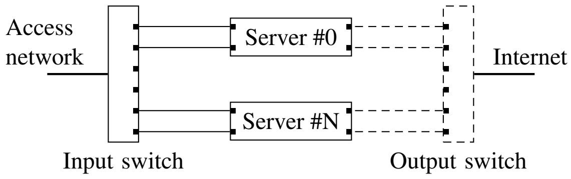


Figure 3: Hardware architecture

3.2 Software Design Principles

Achieving 4.5 Mpps per core or 400 Gbps per server requires an highly efficient implementation: each packet must be processed in less than 500 CPU cycles. We achieve this goal by relying on three design principles that allow an optimal exploitation of the hardware (servers, switches and NICs).

Sharding to the core To enable a high-performance implementation of a software NAT, we completely avoid cross-core data sharing. To this end, in Krononat, we use the CPU core as the unit of sharding in the overall system, thus departing from the traditional per-server or per-NIC sharding scheme. Hence, each CPU core can use its own dedicated connection-tracking table to process the traffic. Each customer is independent and we do not require a global shared connection-tracking table.

To implement this sharding scheme, each CPU core is associated with a NIC and exposed to the network as a distinct entity (i.e., each core has its own dedicated MAC addresses for load balancers to send traffic to it). The load balancers can thus forward the packets to a specific core. Our approach of having a dedicated network entity per core allows a greater control of the traffic-to-core mapping compared to commonly-used hashing-based methods, available on NICs (RSS) or routers (ECMP). This enables (i) a precise control of traffic steering whenever a failed master is replaced by its slave, (ii) sending upstream traffic and downstream traffic, which access the same connection-tracking table, to the same core. This cannot be achieved by Symmetrical RSS [40], because of rewritten IP headers.

To use the hardware as efficiently as possible, we do not dedicate one physical NIC for the input and one to the output for each core. Instead, we use the multi-queue capabilities of NICs (e.g., Intel VMDQ, MACVLAN filter, PCI SR-IOV) to have a dedicated queue for traffic from/to the input switch and a dedicated queue for traffic from/to the output switch[1]. Indeed, residential traffic is highly assymetrical, and dedicated NICs would not evenly use their RX and TX capabilities. Similarly, while a single core can handle the traffic of a 10 Gbps NIC, multiple cores are needed to handle the traffic of 40+ Gbps NICs. Thus, we also use the multi-queue capabilities of the NIC to expose one set of queues/identities per core to implement sharding to the core.

Sharding to the core is therefore highly beneficial for two main reasons: (i) it obviates the need for cross-core synchronization and (ii) it naturally provides NUMA-awareness, as a core never accesses data belonging to another core, and thus only accesses data on his socket.

Switch-based hardware load balancing To handle more than 400 Gbps of traffic, we use several servers. This requires balancing the traffic accross cores and across servers. To balance the load, we rely on the IP routing capabilities[2] of the input/output switches (Figure 3) which (i) remain more efficient than software for routing packets, especially when routing tables are small, (ii) are already present in the system for interconnection. Each shard has its own tunnel endpoint IP, and a dedicated subnet of public IPs. For each shard mapped to a core, a route to this core for the corresponding tunnel endpoint IP is declared to the input switch (to receive upstream traffic); and a route to this core for the corresponding subnet of public IPs is declared to the output switch (to receive downstream traffic). These routes are declared to the switches via BGP. Furthermore, all cores of Krononat thus act as IP routers by having their own IP/MAC addresses and implementing ARP protocol. Our sharding management is detailed in Section 3.4.

This approach avoids dedicating any CPU resources to traffic steering by leveraging existing switches. Also, it allows a more precise traffic steering than hash-based methods (e.g., ECMP). This precise control is needed by stateful network functions (e.g., NAT) that (i) require all packets of a connection to be handled by single core, (ii) have asymmetric headers for upstream and downstream, (iii) require controlling the route after a failover (transitioning from the master to the slave).

No locks on data path Our sharding approach ensures that threads running on different cores can forward or reject traffic without accessing data structures on other cores. This approach removes the need to lock the table to process traffic. Yet, maintenance or fault tolerance traditionally require locking data structures. For fault tolerance, connection-tracking tables need to be copied to other nodes. Traditionally, this is done by locking the table to ensure it is not modified while it is being copied. Because lock acquisition is costly and may block pro-

[1] For the sake of clarity, Figure 4 shows the NIC queues (dedicated MAC and VLAN) to which NAT thread of high-performance nodes are bound rather than the physical network interfaces. Despite the use of a shared hardware infrastructure, we still provide security isolation between the access network and the Internet. To this end, we rely on VLANs (layer 2) and VRF (layer 3) to provide isolated networks such that no packet can be switched/routed directly from the access network to the Internet without going through Krononat. Hence, input/output switch designate the input/output VRF on the physical switch.

[2] All modern entry-level 10/40 Gbps switches offer IP routing capabilities for routing tables of moderate size.

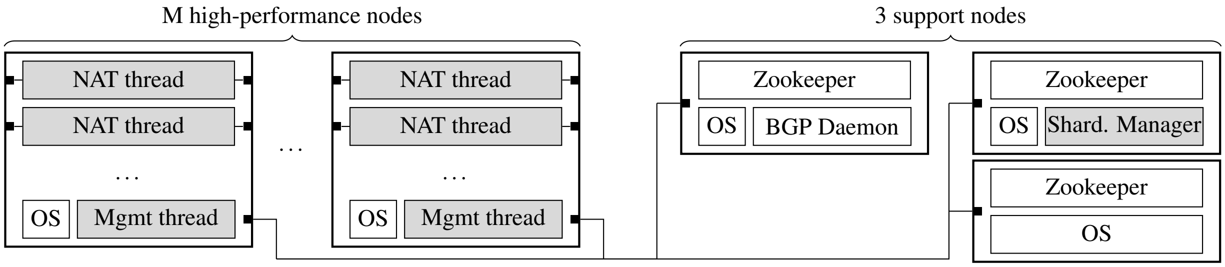


Figure 4: Software architecture

cessing on a given core, locking the table would strongly impair forwarding and is therefore not tractable in our use case. Therefore, we do not use a any lock on the data path. This requires a careful design of data structures and fault tolerance mechanisms, detailed in Section 3.5.

3.3 Software Architecture

We now present how these principles are applied to our system. Krononat comprises software components of its own (Figure 4, gray background) and uses external components (Figure 4, white background). The main functions of each component are detailed below.

NAT Thread The NAT Thread is the central component of Krononat and implements the core functionality of a NAT, as described in Section 2. Each NAT thread is pinned to a CPU core, and has the exclusive use of a set of NIC queues for output and input that it reads in poll-mode. The input switch forwards outbound traffic (GRE-encapsulated) to one input queue of a given NIC and server, depending on the Tunnel destination IP (Figure 2), which is used as the shard identifier (Section 3.2). The NAT thread associated with this input queue receives the traffic, and decapsulates GRE packets. It then forwards traffic to its output queue, and creates entries in its connection-tracking table for new connections. Conversely, each NAT thread receives inbound traffic forwarded by the output switch on its output queue. Inbound traffic is forwarded if and only if it belongs to a connection that has an entry in the table.

Management Thread The management thread communicates with Zookeeper, a strongly-consistent datastore and synchronization service that we use to coordinate all instances of Krononat. It fetches instructions (e.g., master/slave roles for NAT threads) and subscribes to asynchronous events sent by Zookeeper. This cannot be done directly by the NAT threads, because they cannot be interrupted for performance reasons. The management thread is also reponsible for most bookkeeping operations: initialization, statistics collection, etc. It synchronizes with the NAT thread using only non-blocking primitives (i.e., reads and writes to shared memory without locking nor spining).

Sharding Manager To scale to a large number of customers, we divide the load between multiple servers. The sharding manager allocates several shards on each core of each server based on the load of machines and on the traffic. The sharding manager does not directly communicate with the management thread. Instead, it writes the requested shard allocation in Zookeeper. The management thread reacts to updates in Zookeeper and propagates the allocation changes.

Zookeeper In Krononat, Zookeeper is used as central point for storing configuration (shard allocation, network configuration, addressing configuration, routes, etc.) and communicating configuration changes between servers. The use of Zookeeper for storing configuration data greatly simplifies the design of Krononat. For instance, the sharding manager does not need to persist shard allocation by itself. It can be easily restarted, or moved to another server and recover its state from Zookeeper. Similarly, when we start a new Krononat instance to handle more load it can load the global system configuration thanks to Zookeeper. We also use Zookeeper as a distributed lock service, and for detecting server failures.

3.4 Sharding and Fault tolerance

Shard In Krononat, a shard is a fixed-size group of users. In our experiments, we use shards of 256 users, but this number can be adapted. Users in the same shard share the same Tunnel destination IP, but have different Tunnel source IPs (Figure 2). In our experiments, we simulate 16384 users in 64 shards. Each user has a tunnel source IP in the range 172.17.0.0/18, and users belonging to the same shard share an address in the 172.16.0.0/26 range. Users of shard 0 have a source IP in 172.17.0.0-255 and share the tunnel destination IP 172.16.0.0; users of shard 1 have a source IP in 172.17.1.0-255 and share the tunnel destination IP 172.16.0.1; etc. One could also use the 10.0.0.0/8 range to support 4096 shards of 4096 users (i.e., a total of 2^{24} or 16M users).

Shard allocation Based on the traffic in each shard and on the load of each machine, the sharding manager allocates several shards to each NAT thread running on each CPU core of each host. The sharding manager then writes the shard allocation in Zookeeper. In order to steer the traffic to the right core, we leverage IP routing. Each core has its own IP and MAC addresses. Whenever a given core is a master for a given shard, the routing tables are updated so that the given core becomes the default gateway for reaching all subnets associated to the given shard. Each Krononat core thus appear as an independent router in the network. This update is performed by another process that reads the routes in Zookeeper and announces them via BGP to the switches. For instance, if shard 1 has been allocated to a NAT thread (i.e., core) whose input NIC has the IP 172.27.0.1 and output NIC has the IP 127.28.0.1, the BGP will instruct the input switch to route traffic with destination IP 172.16.0.1/32 to 172.27.0.1, and the output switch to route traffic with destination 172.16.1.0/24 to 172.28.0.1. In this way, traffic for shard 1 will be received by the appropriate thread.

Fault tolerance NAT threads store their connection-tracking tables in RAM, which means they will be lost in case of hardware or software failures. One solution would be to persist the connection tracking table to a database. However, this solution would induce a high recovery time. In addition, the database would need to support a very high insertion rate, as each connection establishment results in an insertion. Database systems typically do not support such a high insertion rate because of the consistency and durability guarantees they offer. Instead, we choose to replicate the connection tracking tables in RAM, on another node. More precisely, each NAT thread is declared as *master* for a set of shards, and as *slave* for a distinct set of shards. The master NAT thread for a shard receives the traffic, updates its connection tracking table if necessary and forwards or rejects traffic. The master NAT thread for a given shard also forwards connection tracking table updates to the slave NAT thread of this shard. The slave NAT thread record these tables updates into its own connection tracking table so that entries are replicated. If the server on which the master NAT thread of the shard runs fails, the slave NAT thread becomes the master NAT thread for the shard, and a new slave NAT thread will be assigned for the shard. Server failures are detected using Zookeeper, and shard re-allocations are performed by the sharding manager. We design an ad-hoc replication protocol that allows incremental replication of the connection tracking table without locking it.

3.5 NAT Thread Implementation

The NAT thread continuously polls the NIC queues. To increase efficiency, it processes batches of packets using a run-to-completion model (i.e., packets are not queued except for sending on the network) [1, 33]. It also batches lookups in the connection-tracking table [43, 19].

Hash table Each NAT thread has a single connection-tracking table for all shards it manages either as a master or as a slave. The connection-tracking table is composed of *two* hash tables: one hash table for outgoing traffic and one hash table for incoming traffic. The outgoing traffic hash table maps 6-tuples identifying a connection (customer, protocol; private source ip; private source port; destination ip; destination port) to a 2-tuple (public source ip; public source port) used to rewrite outgoing packets. Symmetrically, the incoming traffic hash table maps 5-tuples (protocol; public source ip; public source port; destination ip; destination port) to a 3-tuple (customer, private source ip; private source port) used to rewrite incoming packets. We use Cuckoo++ hash tables [19] to store the incoming traffic table and the outgoing traffic table. Cuckoo hash tables store an entry at either one of two locations $\mathrm{h}_1(k)$ or $\mathrm{h}_2(k)$, where h_1 and h_2 are two distinct hash functions, and k the key of the entry. Therefore, a key lookup takes at most two memory accesses, allowing Cuckoo hash tables to support a very high lookup rate. This is key requirement in Krononat, as every packet triggers a table lookup. Cuckoo++ hash tables augment Cuckoo hash tables with a small cache-resident bloom filter that avoids checking the second location $\mathrm{h}_2(k)$ in most cases including for negative lookups. This allows Cuckoo++ hash tables to maintain their high performance in presence of large volumes of invalid traffic or Denial-of-Service (DoS) attacks. Furthermore, to support replication without blocking traffic processing in the NAT thread, Cuckoo++ provide specific iterators, that support interleaving of updates and iteration steps by guaranteeing that all entries are iterated over at least once during a full hash-table scan.

Replication As ISPs need to provide uninterrupted Internet access, fault tolerance is a fundamental prerequisite for any network function deployed in their core network. Consequently, Krononat must support single server failures. We achieve this through *replication* of the connection-tracking entries. The master NAT thread for a shard receives all traffic belonging to the shard, and updates its hash tables accordingly. The slave NAT thread of a shard maintains a replica of all connection-tracking entries corresponding to that shard, so that it can take ownership of the shard if the master NAT thread fails. A naive approach for replication would be to lock the hash table on the master NAT thread, and dump the whole data structure on the network. This naive approach has two major defects that make it intractable: (i) locking the hash tables means stopping accepting new connections, which is obviously impossible for availability reasons, (ii) constantly dumping the whole data structure on the network would generate a high replication traffic, and also consume CPU cycles. To address both issues, we design a more elaborate replication protocol that has two modes: (i) *initial replication*, where we transfer the entire contents of the hash tables without locking them thanks to the aforementionned iterators in Cuckoo++, and (ii) *streaming replication*, where the master NAT thread sends incremental updates to the slave NAT thread, so as to reduce network traffic.

Initial replication When a NAT Thread becomes a slave for a shard, it has no knowledge of the connection-tracking entries the master for that shard has: the slave NAT thread needs to receive a full copy of the entries for that shard. This is achieved by the initial replication mode. In this mode, the master NAT thread iterates over hash tables entries, serializes them, and sends them over the network to the slave NAT thread. Note that this initial replication is not performed by an additional thread, it is performed by the NAT thread itself to avoid locking the table. After processing a batch of packets, the master NAT thread iterates over a few entries, and sends them over the network. It then processes the next batch of packets and iterates over the next group of entries. This procedure is repeated until the whole table has been replicated. This interleaving ensures that the initial replication does not preclude packet processing. When a packet is processed, the master NAT thread may update hash table entries (when creating new connections). Therefore, the hash table may be modified in the middle of the initial replication. To support this, Cuckoo++ iterators support modifications. More specifically, we associate a *modification bit* with each table entry. When the iterator sees a table entry, the modification bit is cleared. When a hash table entry is modified, the modification bit is set. The initial replication repeatedly iterates over entries as long as it sees a modified entry to guarantee that no unseen modified entries is left. To accelerate this process, multiple levels of modification bits are used to skip entire groups of unmodified entries.

Streaming replication When the initial replication is completed, the master NAT thread switches to streaming replication mode. In streaming replication mode, whenever it makes a change to the hash tables, the NAT thread inserts a description of this change into a *changelog*. After processing a batch of packets, the master NAT thread extracts remaining entries from the changelog, serializes them and sends them to the slave NAT thread. Because it does not constantly iterate over the hash tables, the streaming replication mode uses less CPU cycles and less network bandwidth than the initial replication mode. The replication protocol uses acknowledgments and retransmissions and in case of failure the slave is declared out of sync and must go through initial replication again.

Server CPU	Krononat (10G port)	Traffic gen.
1x E5-2695v3	1	2
1x E5-2695v3	3	2
1x E5-2643v3	4	0
2x E5-2690v4	4	4

Table 3: Allocation of servers in our testbed

4 Evaluation

Implementation Krononat (see Fig 4) is implemented in C (30K lines – gcc 5.4) on top of DPDK 17.08. The sharding manager is implemented in Scala (2K lines). The BGP part consists of a 500 lines wrapper between ZooKeeper 3.4.8 and GoBGP 1.18.

Hardware Our hardware testbed consists of four Dell R730 servers with varying CPU configurations (Table 3). They are configured in performance mode, with Turboboost disabled and equipped with Intel X540 10Gbps dual-port NICs. They are interconnected by an entry-level 10Gbps Alcatel OS6900-T20 switch, which is configured to operate as an IP router with multiple VLANs and VRFs (Virtual Routing Functions), so as to provide isolation between access, Internet, and management networks. Our testbed uses addresses in the range 172.16.0.0/12 so as not to conflict with the enterprise networks. This slightly limits the range of some parameters but our experiments show that those parameters have very little impact on performance anyway (Subsection 4.1). The hardware of the testbed is shared between Krononat and a traffic generator that we designed. Table 3 shows the mapping of 10G NIC ports.

Traffic Generator Testing the limits of Krononat requires generating a very large amount of traffic (tens of Mpps). We did not find a traffic generator that is able to generate such a load by utilizing several nodes, so we designed our own traffic generator to test Krononat. Our traffic generator builds upon DPDK, similarly to MoonGen [10] or pkt-gen [39], and borrows principles from Krononat such as (i) share-nothing, (ii) sharding to the core, (iii) switch-based hardware load balancing. It generates stateful traffic and keeps track of established connections. Each 10 Gbps NIC allocated is managed by a group of 4 threads (RX/TX Access/Internet). All instances (1 per server) emulate independant users and synchronize using Zookeeper (results, parameters, phase). This scale-out design allows a close to linear scalability and generating traffic beyond the scale of one server.

Metrics In our evaluation, we measure:

Rate for Connection initialization correspond to the rate at which packets (UDP) of new flows/connections are processed (100% upstream traffic). On actual Internet traffic, connection initialization packets represent 1-5% of the traffic [37, 18, 24, 5, 38, 31].

Rate for Established connections is the rate at which packets (UDP) belonging to existing flows are processed by the system. Our objective is to exceed 4.5 Mpps. We

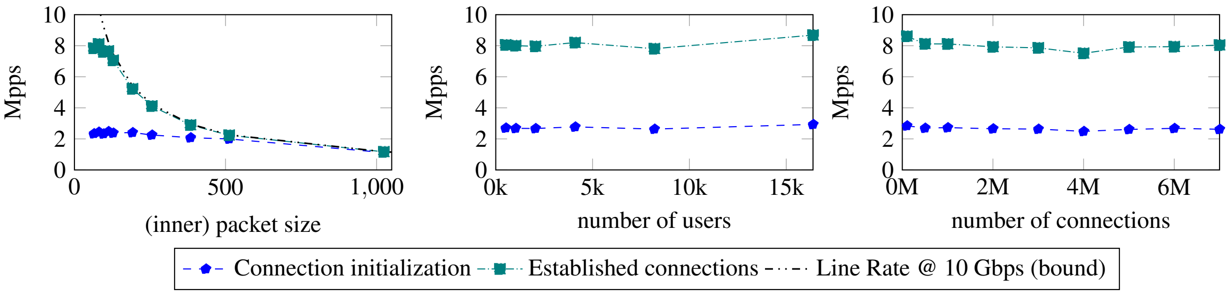


Figure 5: Performance impact of parameters in single-core case for Krononat (raw performance) (1 server)

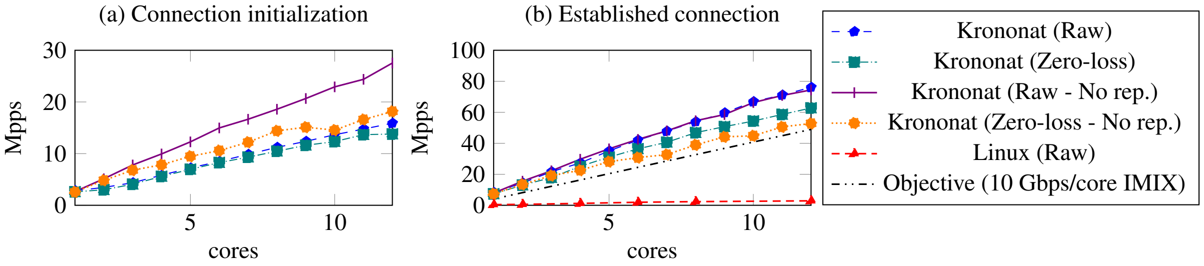


Figure 6: Performance (Mpps) in the multi-core case (4 servers).

measure for 50% upstream and 50% downstream traffic.

They are reported [7] as: (i) *raw* is the rate of traffic going through the system while flooding it; (ii) *zero-loss* is the rate which can be sustained without losing packets during 15 seconds. The *zero-loss* rate is only relevant for systems with real-time guarantees/non-blocking implementations (i.e., not for Linux-based NAT).

We also evaluate ***the duration of service interruption*** whenever a server leaves the system or crashes.

4.1 Influence of parameters

First, we evaluate the impact of a few parameters on Krononat performance. We report the raw rate for both connection initialization and established connections on Figure 5. Performance is stable at approximately 8 Mpps for established connections regardless of the number of users or connections. The performance is reduced as the packet size is increased: this is because the system achieves line rate (10 Gbps) and is therefore NIC-bound rather than CPU-bound. This shows the high efficiency of Krononat.

As we have shown that the packet size, number of users and number of connections have little to no impact on performance, we use fixed values in the remainder of this section: (i) 64-bytes packets, (ii) 16k users, (iii) 5M connections per core. We use the lowest packet-size (64 bytes without GRE encapsulation) to remain CPU-bound since we aim at evaluating the CPU-efficiency. This also allows our traffic generator, which is allocated only 8 NIC ports, to generate as many packets as necessary to evaluate Krononat on 12 cores without being limited by the speed of network interfaces.

4.2 Scalability

We benchmark Krononat with multiple cores, on all 4 testbed servers. Krononat is evaluated both with and without replication to show the overhead of replication. To give an idea of the achieved performance, we also plot performance of a trivial NAT system that uses the Linux kernel implementation and Linux namespaces[3]. We also display our performance objective: 400 Gbps/server or 4.5 Mpps per core, which enables the use of dense servers (i.e., fitted with as many NICs as CPUs support).

The results are reported on Figure 6 using 1 to 12 cores, averaged over 10 runs with random placement of NAT threads on our 4 servers. Krononat on 12 cores achieves 15 million connection initialization per second, with replication enabled (halved from Krononat without replication). For established connections, Krononat with replication is able to process packets at 77 Mpps on 12 cores[4]. The penalty when measuring performance with the zero-loss constraint is limited.

Kronont offers much higher performance than the Linux-based NAT. For established connections, Linux only achieves 0.6 Mpps on 1 core and scales to 2.9 Mpps on 12 cores. This is because Linux is a general-purpose system not dedicated to multi-tenant NAT; thus its design

[3] To maximize Linux performance, we take care to avoid extreme settings, and limit the experiment to 32 users (i.e., 32 namespaces) and 50000 flows. We distribute traffic accross cores of our Intel Xeon E5-2690v4 using RSS in a NUMA-aware way (i.e., on cores on the same socket as the NIC). Despite these advantageous settings, Linux performance remains low compared to Krononat.

[4] The performance for established connections without replication is lower as in this case, the sharding manager is not able to rebalance the load by swapping roles between master and slave cores.

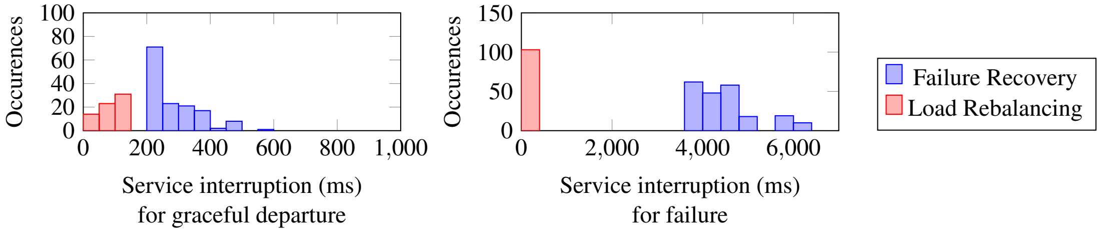


Figure 7: Service interruption due to failure or departure injection (4 servers)

favors configurability and generality over performance. This shows that Linux is unusable as an ISP-grade NAT solution. This is even more salient considering that Krononat also provides by default distribution accross servers and fault-tolerance. Krononat performance is well above our objective, reaching 6.3 Mpps/core when run on 12 cores, with replication enabled. These experiments show the excellent scalability obtained thanks to our scale-out architecture, consistently with the scalability of the underlying hash-table [19].

Running such experiments proved challenging: (i) network requirements inhibits the use of the cloud; (ii) commodity networking remains relatively expensive for large-scale experiments, (iii) simulation or virtualization overhead would make performance evaluation of such DPDK-based implementation irrelevant. We were thus limited by the networking equipment that we shared between traffic generation and Krononat. Nonetheless, our experiments involve four real servers and up to 12 cores, reaching up to 76 Mpps (6.3 Mpps/core), which is well above our target of 4.5 Mpps/core. The scale of these experiments also show the interest of using sharding to the core and hardware-based traffic steering to implement stateful multi-core multi-server traffic generators.

4.3 Service interruption duration

Beyond its high-performance and scalability, a major feature of Krononat is fault tolerance. As the state for each shard is continuously replicated on a master and a slave core, Krononat can recover from a server failure without disrupting service. Replication is also useful for dynamically adapting the number of servers (e.g., graceful departure) and rebalancing the load (i.e., swapping master and slave). We inject both graceful departures and hard failures and measure the durations of service interruptions in both cases. We report the durations of service interruptions due to recovery (i.e., the slave replaces the departed master – red) and to load-rebalancing (i.e., master and slave swap their roles – blue) on Figure 7.

Service interruptions due to graceful departure (i.e., the server disconnects gracefully from Zookeeper) remain below 500ms. This corresponds to the delay for the sharding manager to compute a new allocation, which is applied to NAT thread; and to announce routes via BGP. In the case of hard failures (i.e., the server does not disconnect gracefully from Zookeeper), the detection is left to ZooKeeper heartbeat. This increases the recovery delay to 4-7 seconds. This recovery is automatic, without any human intervention, ensuring that the system is highly available. Finally, interruptions due to load rebalancing last less than 100ms. Indeed, to rebalance the load, the sharding manager swaps roles between some masters and slaves. In this case, the interruption is mainly the delay for BGP to apply the new routes.

In all cases, these durations are low enough so that clients retransmit lost packets without declaring connections dead. This ensures that end users are not impacted. We performed real-life experiments by redirecting our own Internet traffic through Krononat for a few hours. The interruptions due to injected failures remained unnoticeable in web browsing and video streaming usages. Indeed, short interruptions are hidden by software buffering or retransmission, thus avoiding user-visible errors.

4.4 Recovery from failure

To further illustrate the system reaction to a crash, we report how Krononat (4 NAT threads on 4 servers) reacts step-by-step when one of the servers is electrically powered down. An electrical failure is triggered at the 7th second. For a short duration, approximately 16 shards have lost their master, and 13 have lost their slave. The service is thus interrupted for 16 shards. The system becomes fully available again within 200ms (i.e., no more shards in slave only mode). Shards that lost their slave or master are allocated a new slave that is being initialized (20th to 35th second). This initialization generates limited replication traffic ($<$ 65 Mbps) and has a very limited impact on the performance: traffic is still processed at approximately 24 Mpps. The performance drops slightly during recovery because the master must read and transmit its state to the new slave, and the new slave must record this state. The performance drop is limited thanks to the absence of locking of the connection-tracking table allowed by our replication protocol that allows initial replication to occur without freezing state. Without any human operator intervention, at the 35th second, the system becomes fully tolerant to failures again: all shards have a master and a synchronized slave.

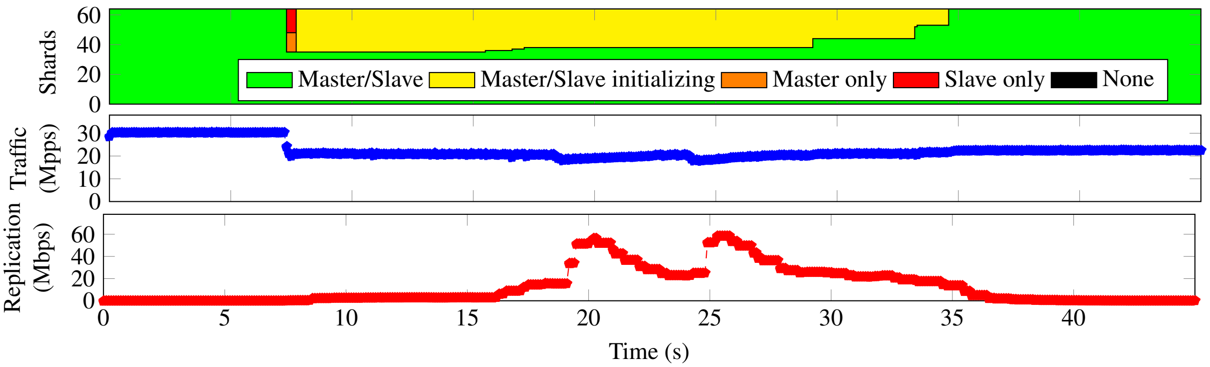


Figure 8: Execution on a system with 4 workers, on 4 servers. A server crash is introduced at time t=7s.

5 Discussion

SDN For load-balacing, Krononat relies on IP routing configured through BGP, and ARP/MAC learning for the discovery of Krononat instances. This greatly simplified the implementation of hardware-based load balancing. This choice also allows using VRF-based isolation, as well as easy inspection of routing table (e.g., using standard switch user interface). Furthermore, many production-ready BGP libraries are available.

Openflow [23] or P4 [6] could have been an alternative, but it requires specific models of switches. Also, the configuration or capabilities of switches for OpenFlow are not always well documented. Note that BGP requires routing based on IP addresses : OpenFlow could thus be advantageous if we want to do traffic steering based on other criteria (e.g., MAC addresses) or if non-IP protocols were used for tunneling on the access side.

Unavailability delay Krononat relies on Zookeeper for failure detection. This leads to uncompressible delays for recovery (4-7s) mainly due to Zookeeper failure detection. To improve this, one could rely on BFD [17] to monitor links at the switches and declare in BGP a primary route to the master and a secondary route to the slave. This way, in case of server or link failure, the switch could immediately react and route packets to the slave without waiting for Zookeeper to detect the failure. Yet, the unavailabilities we observe remain un-noticed in practice with typical web traffic including live streaming, mostly hidden by buffering and TCP retransmission.

6 Related Work

Krononat applies techniques such as kernel-bypass, run-to-completion, and core pinning [33, 1, 4, 8, 20] which are necessary to achieve high-performance on modern processors. Krononat shows how to put these in practice when dealing with a mutable state by relying on sharding-to-the-core to achieve share-nothing.

Switching or routing NFVs, targetting COTS servers, have already been studied and implemented [8, 30, 22]. ClickOS [22] also considers advanced middleboxes such as BRAS and CG-NAT but achieves only 2.3 Mpps with a 4-core processor. Switching and routing NFVs rely on a *small and static* state. They can thus share state between cores with limited performance penalty, which is not tractable in connection-tracking systems. Our papers extends the share-nothing principle beyond NIC queues and details how to distribute traffic to cores with finer control than classical hash-based traffic distribution.

NFV frameworks [28, 27, 42, 15, 41] consider the communication between NFV functions. They show that context switches have huge overhead and either (i) avoid containers/VMs by using other types of isolation [28, 42], (ii) optimize communication between containers/VMs [14, 15, 30]. They provide capabilities to share resources between multiple VMs running services consuming only a fraction of the resources. In Krononat, we do not need to isolate several chained components of our datapath, nor to share server resources between multiple small applications. Thus, these frameworks do not fit our use case and add complexity for little benefits.

Virtual switches [15, 30], can help in providing features missing from hardware switches or NICs. In our case, rather than providing software-based traffic distribution to address limitations of hardware NIC and switches, we choose to design our sharding scheme (e.g., using IP routing features) so that it can be supported by entry-level 10 Gbps switches and commodity 10 Gbps NICs. To avoid context switches, we use a single process and the run-to-completion model similarly to NetBricks [28] or BESS [15]. Yet, the modularity they bring comes at a cost: during prototyping we noticed that dynamic dispatch used in BESS or NetBricks can have a non-negligible overhead compared to static dispatch as it prevents some compiler optimizations.

The aforementionned frameworks focus on directing packets within a server, while Krononat provides a solution for directing packets accross servers through switch-based load balancing. Also, these frameworks [15, 28] have limited features for directing packets for both flow directions to a single core (e.g., tunneled traffic, rewritten headers) as they rely on hash-based distribution (e.g., RSS), or attach a thread per NIC. By using MAC addresses to direct packet to per-core queues, we borrow from SoftBricks [8], an interesting paper that considers distribution accross server but for routing and VPN applications. It features an inspiring description of hardware capabilities and their impact on software router design.

Interestingly, designing for multi-tenancy can impact efficiency. CORD vSG [2] relies on namespace per user. This leads to one queue or datastructure per user. It prevents batching, which is yet key to high performance in large hash tables [19]. Indeed, a received batch is unlikely to contain only packets for a single user. On the contrary, Krononat relies heavily on batching to achieve its performance objective as commonly practiced in high-performance networking [1, 33].

Systems tracking connections such as load-balancers [9, 13, 29] or NAT/FW [16] also deal with the difficulty of preserving mutable state. A first approach is to rely on an external reliable database such as Memcached [11], RamCloud [26] or Adhoc [13]; while simplifying design, this comes with the cost of running the database (additional servers) and accessing it (dedicated NICs consuming PCIe lanes). A second approach is to rely on consistent-hashing, like the Maglev load balancer [9]. One enabler for this approach is that Maglev handles only unidirectionnal traffic as reverse traffic relies on DSR (Direct Server Return). MagLev achieves 2-3 Mpps/core. Krononat tackles a more challenging use case than Maglev (NAT versus load balancing), which requires handling bidirectional traffic. The NAT in [16] achieves 5 Mpps on 12 cores using RAMCloud. Krononat largely exceeds the performance of [16]. This is because Krononat underlying hash table is much faster (e.g., 10M insertions/second/core, 35M lookups/second/core and 350M lookups/second for 12 cores) than a remote RAMCloud (0.7M insertion/second and 4.7M lookups/second on 12 cores [16]). In addition, remote database accesses prevent run-to-completion.

Despite not relying on an external database, Krononat still offers reliability as it passively replicates of all connection entries. This design allows Krononat to offer a much higher performance than [16], strongly reducing costs for ISPs. An alternative design [34] to using a reliable databases is to snapshot the NFV periodically and log all packets to allow restarting the NFV from a snapshot and replaying traffic if needed. Interestingly, this approach allows adding reliability to any middlebox with little to no modification. Yet, this also comes at the cost of performance as FTMB is limited to 6 Mpps using 16 cores. Overall, Krononat favors liveness and performance over strong consistency. Indeed, for networking applications, strong consistency has a high performance impact, and may even be undesirable. A common choice in databases is to block or delay updates if they cannot be durably recorded, but for networking this means dropping any new connection thus interrupting the service. An alternative choice is thus to favor liveness: in the rather unlikely event of simultaneous failure of two servers, software clients will re-establish connections, causing little trouble.

Load-balancers [9, 29] often rely on IP routing as a first layer for traffic distribution from the Internet. Each load-balancer owns one or several of the VIP (virtual IPs) to capture traffic from the Internet. This is similar in design to our Tunnel end point IP addresses. Krononat further exploit this to also capture reverse traffic and use a different granularity by sharding down to the core-level so as to allow an highly efficient implementation.

Interesting concurrent work by, Araujo et al [3], describes a load-balancer design that shares a few key observation with Krononat : (i) commodity switches are incredibly efficient at distributing packets, both papers thus offload as much of their work as possible onto the switch, (ii) doing so requires to adapt the sharding and the network addressing scheme so that it is supported by commodity switches. Yet, as we target different applications (i.e., NAT and stateful firewall for us and load-balancing for them), other points of the design differ (e.g., permanent replication vs on-demand draining, handling bi-directional traffic, updating the routing table rather than updating the ARP table).

7 Conclusion

We presented Krononat, a high performance stateful networking service providing NAT and firewall for large-scale residential access network of ISPs. Krononat has a close to linear scalability thanks to its design relying on sharding to the core, and was shown to handle 77 Mpps on 12 cores, fully exploiting our testbed. It is designed for scale-out both accross cores and accross servers; it should scale linearly well beyond 12 cores and 4 servers.

Our design relies on sharding to the core, by exposing each core as an independent entity on the network. This allows traffic steering accross cores and servers to be performed by the switches freeing precious CPU resources. Traffic steering is based on IP routing as it allows a fine control, useful for stateful NFV functions for which RSS/ECMP offer insufficient control. Beyond Krononat, these principles proved useful for building the scale-out traffic generators that we use for the performance evaluation. We believe these principles can also apply widely to high-performance implementations of stateful NFV functions.

References

[1] DPDK: Data Plane Development Kit. `http://dpdk.org`.

[2] Residential CORD. `https://wiki.opencord.org/pages/viewpage.action?pageId=1278090`.

[3] ARAUJO, J. T., SAINO, L., BUYTENHEK, L., AND LANDA, R. Balancing on the edge: Transport affinity without network state. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, 2018), NSDI'18, USENIX Association, pp. 111–124.

[4] BARBETTE, T., SOLDANI, C., AND MATHY, L. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (Washington, DC, USA, 2015), ANCS '15, IEEE Computer Society, pp. 5–16.

[5] BOCCHI, E., KHATOUNI, A. S., TRAVERSO, S., FINAMORE, A., MUNAFÒ, M., MELLIA, M., AND ROSSI, D. Statistical network monitoring: Methodology and application to carrier-grade nat. *Computer Networks 107* (2016), 20 – 35. Machine learning, data mining and Big Data frameworks for network monitoring and troubleshooting.

[6] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[7] BRADNER, S. Benchmarking terminology for network interconnection devices. IEETF RFC 1242, 1991.

[8] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 15–28.

[9] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 523–535.

[10] EMMERICH, P., GALLENMÜLLER, S., RAUMER, D., WOHLFART, F., AND CARLE, G. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference* (New York, NY, USA, 2015), IMC '15, ACM, pp. 275–287.

[11] FITZPATRICK, B. Distributed caching with memcached. *Linux J. 2004*, 124 (Aug. 2004), 5–.

[12] FUKUDA, K., CHO, K., AND ESAKI, H. The impact of residential broadband traffic on japanese isp backbones. *SIGCOMM Comput. Commun. Rev. 35*, 1 (Jan. 2005), 15–22.

[13] GANDHI, R., HU, Y. C., AND ZHANG, M. Yoda: A highly available layer-7 load balancer. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 21:1–21:16.

[14] GARZARELLA, S., LETTIERI, G., AND RIZZO, L. Virtual device passthrough for high speed vm networking. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (Washington, DC, USA, 2015), ANCS '15, IEEE Computer Society, pp. 99–110.

[15] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. Softnic: A software nic to augment hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[16] KABLAN, M., ALSUDAIS, A., KELLER, E., AND LE, F. Stateless network functions: Breaking the tight coupling of state and processing. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2017), NSDI'17, USENIX Association, pp. 97–112.

[17] KARTZ, D., AND WARD, D. Bidirectional forwarding detection (bfd). IETF RFC 5880, 2010.

[18] KIM, M.-S., WON, Y. J., AND HONG, J. W. Characteristic analysis of internet traffic from the perspective of flows. *Comput. Commun. 29*, 10 (June 2006), 1639–1652.

[19] LE SCOUARNEC, N. Cuckoo++ Hash Tables: High-Performance Hash Tables for Networking Applications. *ArXiv e-prints* (Dec. 2017).

[20] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 429–444.

[21] MAIER, G., FELDMANN, A., PAXSON, V., AND ALLMAN, M. On dominant characteristics of residential broadband internet traffic. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2009), IMC '09, ACM, pp. 90–102.

[22] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 459–473.

[23] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 69–74.

[24] MUSCARIELLO, L. *On Internet Traffic Measurements, Characterization and Modelling*. PhD thesis, Politecnico Di Torino, 2006.

[25] NEVILLE-NEIL, G., AND THOMPSON, J. Measure twice, code once: Network performance analysis for freebsd. In *ASIA BSD Conference* (2015).

[26] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The ramcloud storage system. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[27] PALKAR, S., LAN, C., HAN, S., JANG, K., PANDA, A., RATNASAMY, S., RIZZO, L., AND SHENKER, S. E2: A framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 121–136.

[28] PANDA, A., HAN, S., JANG, K., WALLS, M., RATNASAMY, S., AND SHENKER, S. Netbricks: Taking the v out of nfv. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 203–216.

[29] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 207–218.

[30] Pfaff, B., Pettit, J., Koponen, T., Jackson, E. J., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., and Casado, M. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 117–130.

[31] Qian, L., and Carpenter, B. E. A flow-based performance analysis of tcp and tcp applications. In *2012 18th IEEE International Conference on Networks (ICON)* (Dec 2012), pp. 41–45.

[32] Quan, L., Heidemann, J., and Pradkin, Y. When the internet sleeps: Correlating diurnal networks with external factors. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, ACM, pp. 87–100.

[33] Rizzo, L. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 9–9.

[34] Sherry, J., Gao, P. X., Basu, S., Panda, A., Krishnamurthy, A., Maciocco, C., Manesh, M., Martins, J. a., Ratnasamy, S., Rizzo, L., and Shenker, S. Rollback-recovery for middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 227–240.

[35] Srisuresh, P., and Egevang, K. Traditional ip network address translator (traditional nat). IETF RFC 3022, 2001.

[36] Strowes, S. D. Diurnal and weekly cycles in ipv6 traffic. In *Proceedings of the 2016 Applied Networking Research Workshop* (New York, NY, USA, 2016), ANRW '16, ACM, pp. 65–67.

[37] Thompson, K., Miller, G. J., and Wilder, R. Wide-area internet traffic patterns and characteristics. *Netwrk. Mag. of Global Internetwkg. 11*, 6 (Nov. 1997), 10–23.

[38] Velan, P., Medkova, J., Jirsik, T., and Celeda, P. Network traffic characterisation using flow-based statistics. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium* (April 2016), pp. 907–912.

[39] Wiles, K. pktgen-dpdk. `http://pktgen-dpdk.readthedocs.io/`.

[40] Woo, S., and Park, K. Scalable TCP Session Monitoring with Symmetric Receive-side Scaling. Tech. rep., KAIST, 2012.

[41] Woo, S., Sherry, J., Han, S., Moon, S., Ratnasamy, S., and Shenker, S. Elastic scaling of stateful network functions. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, 2018), NSDI'18, USENIX Association, pp. 299–312.

[42] Zhang, W., Liu, G., Zhang, W., Shah, N., Lopreiato, P., Todeschi, G., Ramakrishnan, K., and Wood, T. Opennetvm: A platform for high performance network service chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (New York, NY, USA, 2016), HotMIddlebox '16, ACM, pp. 26–31.

[43] Zhou, D., Fan, B., Lim, H., Kaminsky, M., and Andersen, D. G. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2013), CoNEXT '13, ACM, pp. 97–108.

Accurate Timeout Detection Despite Arbitrary Processing Delays

Sixiang Ma
The Ohio State University

Yang Wang
The Ohio State University

Abstract

Timeout is widely used for failure detection. This paper proposes SafeTimer, a mechanism to enhance existing timeout detection protocols to tolerate long delays in the OS and the application: at the heartbeat receiver, SafeTimer checks whether there are any pending heartbeats before reporting a failure; at the heartbeat sender, SafeTimer blocks the sender if it cannot send out heartbeats in time. We have proved that SafeTimer can prevent false failure report despite arbitrary delays in the OS and the application. This property allows existing protocols to relax their timing assumptions and use a shorter timeout interval for faster failure detection. Our evaluation shows that the overhead of SafeTimer is small and applying SafeTimer to existing systems is easy.

1 Introduction

This paper presents SafeTimer, a mechanism to enhance existing timeout detection protocols to prevent false failure reports caused by long delays in the OS and the application. With the help of SafeTimer, existing protocols can relax their timing assumptions and thus use a shorter timeout interval for faster failure detection.

Timeout is widely used in distributed systems to detect failures [1, 6, 13, 24, 29, 45]: a node periodically sends a heartbeat packet to others and if the receiver does not receive the heartbeat in time, it may report a failure and may take actions to recover the failure.

Although this idea is simple, delays of packet transfer create a problem: if a receiver misses a heartbeat, is it because the sender has not sent the heartbeat, which indicates a failure, or is it because the heartbeat is delayed somewhere, which should not indicate a failure?

To address this problem, existing systems use one of the following approaches: the first is to prevent false failure reports by setting an appropriate timeout interval. However, such setting requires certain timing assumptions about the communication channel [4, 5, 18] and creates a dilemma: on one hand, these assumptions should be conservative enough to tolerate abnormal events that can cause long delays (e.g., congestion), which means the timeout interval should be long. On the other hand, long timeout interval can hurt system availability, because the system has to wait for a long time before recovering the failure. A recent study shows that inappropriate timeout interval is a major cause of timeout related bugs, leading to various problems like data loss or system hanging [19]. The second approach is to ensure correctness despite false failure reports, using protocols like Paxos [34, 35, 42]. This approach allows short timeout for better availability, but its cost is usually higher.

SafeTimer enhances the first approach to tolerate a subset of those abnormal events, without requiring any timing assumptions. It thus allows existing protocols to relax their timing assumptions to use a shorter timeout interval, without sacrificing the accuracy of timeout detection. It is motivated by two insights.

First, conservative assumptions are only necessary if the communication channel is a blackbox, which cannot provide any additional information other than receiving a packet. If the channel can tell whether a packet is pending or dropped, the receiver can simply check whether there is a pending or dropped heartbeat when missing a heartbeat. This approach can prevent false failure reports without requiring any timing assumptions.

Second, we observe that modeling the whole communication channel as a blackbox is too pessimistic: the routing layer usually does not provide the users with information like packet drops, so it is reasonable to model routing as a blackbox; the OS and the application, however, can provide precise information about its packet processing and thus could be modeled as a whitebox. Furthermore, in today's datacenters, the whitebox part often incurs delays that are comparable to or even larger than those of the blackbox part: on one hand, intra-datacenter networking delays usually range from tens of microseconds to a few milliseconds and can be further reduced to hundreds of nanoseconds with techniques like Infiniband [31]. Improvement in bandwidth and protocols [14, 46] have significantly reduced the chances of packet drops. On the other hand, a traditional OS can delay processing by several milliseconds because of time sharing or page fault, etc. Such delay can occasionally grow to several seconds for reasons like SSD garbage collection [32] and can grow even higher in abnormal cases (see Section 2).

Because of these two insights—1) the delay of the whitebox part is significant among communication and 2) there exist more effective solutions for the whitebox part—SafeTimer naturally uses a more effective solution for the whitebox part; for the blackbox part, SafeTimer relies on existing protocols and their assumptions.

At the receiver side, SafeTimer guarantees that *as long as the network interface card (NIC) has either delivered or dropped the heartbeat before the deadline, the receiver will not report a failure.* To achieve this property, SafeTimer's receiver module checks whether there are any pending or dropped heartbeats in the system before reporting a failure. Implementing this idea, however, is challenging, because modern OS incorporates a highly concurrent pipeline for fast packet processing. Naive solutions like pausing all its threads requires an intrusive modification to kernel, which is undesirable.

To solve this problem, we propose a non-blocking solution: when the timer expires at t, SafeTimer's receiver module will send a barrier packet to itself. By crafting the barrier packet and configuring the OS properly, SafeTimer ensures that if the receiver module receives the barrier, all heartbeats processed by the NIC before t must have been either delivered to the application or dropped. Therefore, if the receiver module has neither received the heartbeat nor observed any packet drops, it can safely report a failure.

At the sender side, SafeTimer guarantees that *if the sender has not sent out a heartbeat in time, the sender will not be able to send out any new packets.* Such suicide idea is not novel [8, 22], but previous solutions that actively kill or reboot the sender do not work when considering long processing delays, because the kill or reboot operations may be delayed as well, leaving the sender alive. To solve this problem, SafeTimer incorporates a passive design: SafeTimer's sender module maintains a timestamp to identify till when it is valid for the sender to send new packets. The sender module updates this timestamp when successfully sending a heartbeat and checks this timestamp before sending any packets. By doing so, SafeTimer prevents a sender which fails to send heartbeat in time to affect other nodes in the system.

One can enhance an existing timeout detection protocol by applying SafeTimer at both the sender and the receiver. We can prove that, as long as the existing protocol's assumptions about the blackbox part hold, SafeTimer is accurate (i.e., never report failure for a correct sender) despite arbitrary delays in the whitebox part and is complete (i.e., eventually report failure for a failed sender) when the receiver does not experience slow processing or packet drops for sufficiently long [12]. Such properties indicate that one does not need to make conservative assumptions about the whitebox part, and thus can use a shorter timeout interval to improve availability.

Our evaluation shows that the overhead of SafeTimer is negligible when processing big packets and at most 2.7% when processing small packets; SafeTimer can prevent false failure reports when long processing delays are injected; and applying SafeTimer to HDFS [27, 45] and Ceph [9] is easy.

2 Motivation

2.1 Long delays in OS and application

SafeTimer allows existing timeout detection protocols to relax their timing assumptions by excluding delays in the OS and the application. To demonstrate the potential benefits of such relaxation, we present a number of abnormal events that can cause long delays.

- **Disk access.** Disk accesses caused by logging heartbeats [29, 45] or page faults can block heartbeat processing. A typical hard drive has an average latency of tens of milliseconds and an SSD usually has a lower average latency. Worst-case latency, however, is much longer: SSD's internal garbage collection can delay an access by more than one second [32]. Our experiment with hard drives shows that when processing frequent random writes, the buffering mechanism in the file system can occasionally introduce a latency of tens of seconds, when it flushes many random writes.
- **Packet processing.** OS kernel can drop packets at different layers when it runs out of buffer space, which can cause extra delay. Furthermore, handling of abnormal packets may cause a significant delay as well. For example, when Linux receives a packet to an unopened port, it will report "port unreachable" to the router using ICMP [30]. In our experiment, a large number of such abnormal packets can delay the processing of heartbeat by more than two seconds.
- **JVM garbage collection.** Garbage collection in a Java Virtual Machine (JVM) can block the execution of the application. Our experiment on a JVM with 32GB of memory shows that when the memory is close to be fully utilized, a single garbage collection can take up to 26 seconds, even when using parallel GC. A recent survey [19] has observed similar problems in ZooKeeper and HBase (HBase-3273 [26]).

- **Applicaton specific delays.** Applications may have specific logics that can cause long delays occasionally. For example, previous works have reported that HDFS DataNode's heartbeat sending thread may be blocked by the task of scanning local data, which could take long [48]. Although newer versions of HDFS have fixed this problem, our investigation shows that similar problems still exist: the heartbeat sending thread can also be blocked by the task of deleting directories, which can take long as well. A similar problem has been reported in Ceph, in which a heavy rejoin operation can block heartbeat processing [11].

As shown in these examples, some events in the OS and the application can cause delays of tens of seconds, which are comparable to or larger than many systems' default timeout intervals (e.g., 30 seconds in HDFS [28], 5 seconds in ZooKeeper [25], 20 seconds in Ceph [10]). Furthermore, some of these delays may grow longer if a machine has more resource (e.g., more memory for JVM garbage collection).

Existing timeout detection protocols must make their timing assumptions conservative enough to cover all the events mentioned above. For example, to tolerate long garbage collection in ZooKeeper [26], the developers increased their timeout intervals, which will hurt system availability as discussed previously. With the help of SafeTimer, however, they can tolerate these events without requiring any timing assumptions, and thus can use a shorter timeout for faster failure detection.

2.2 Can we provide timing guarantees?

The above problems would be trivial if the OS and the application can provide hard real-time guarantees for heartbeat processing, but during our failed attempts, we find this is a challenging task on commodity OS.

Isolated resource for heartbeats. To prevent other tasks from interfering with heartbeat processing, the application can reserve resources (e.g., a socket) for heartbeat processing. However, this approach cannot prevent such interference in OS kernel. For example, packets from different sockets can be handled by the same thread or CPU core in the kernel; even if heartbeat handling does not need to make disk I/Os, page fault in the kernel may incur a disk I/O, blocking heartbeat processing.

Processing heartbeats at lower layers. To avoid delays in the OS kernel, one can implement heartbeat sending and checking at lower layers, as close as possible to the NIC. This approach can avoid many types of delays, but cannot eliminate them, because heartbeat checking can only happen after the OS reads a packet, which means delays in handling interrupts and reading packets can still cause false failure reports.

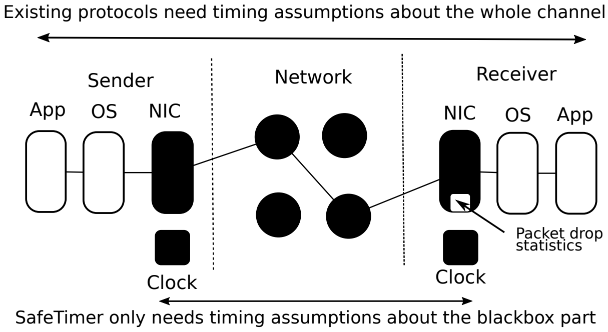

Existing protocols need timing assumptions about the whole channel
Sender
Network
Receiver
App
OS
NIC
NIC
OS
App
Packet drop statistics
Clock
Clock
SafeTimer only needs timing assumptions about the blackbox part

Figure 1: System model: SafeTimer can tolerate long delays in the whitebox part without timing assumptions.

Real-time OS. Real-time Linux [43] and other real-time frameworks for Linux such as RTAI [44] and Xenomai [49] can give higher priority to certain interrupts, so that they wouldn't be delayed by other interrupts. However, this approach can only guarantee an interrupt handler is triggered in time, but cannot guarantee when the OS can finish reading a packet. The latter requires us to analyze the worst-case execution time of handling interrupts and reading packets, which is a challenging task on complicated kernel code with frequent synchronizations.

We find these approaches, even combined, cannot achieve hard timing guarantees for heartbeat processing. The fundamental problem is that commodity OSes are designed with the principles of resource sharing and high concurrency, which is against the goal of strict timing guarantees. Therefore, finally we give up the attempts to provide timing guarantees. Instead, we investigate whether we can prevent false failure reports **assuming delays in the OS and the application can be arbitrary**.

3 Model

The goal of SafeTimer is to enhance existing timeout detection protocols to tolerate long processing delays in the OS and the application. To achieve this goal, SafeTimer makes a few assumptions about the existing protocol: at the receiver side, SafeTimer assumes the receiver defines multiple time intervals and reports a failure if it does not receive any heartbeats during an interval. At the sender side, SafeTimer assumes the application has its own rules to decide when to send heartbeats and whether heartbeats are sent successfully, based on its timing assumptions. Furthermore, SafeTimer assumes these intervals and assumptions are configurable, so that the user can use a shorter timeout interval with the help of SafeTimer.

SafeTimer enhances existing protocols to tolerate a subset of abnormal events without requiring timing assumptions. Figure 1 shows which events SafeTimer can tolerate: the blackbox part includes the network interface cards (NICs) at both sides, the clocks at both sides, and packet routing between two NICs; the whitebox part includes the OS and the application's logic to process packets at both sides. SafeTimer can tolerate long delays in the whitebox part without requiring any timing assumptions. Instead, SafeTimer only assumes that, a node will eventually finish processing a heartbeat and SafeTimer can observe the result (either delivered or dropped). For the blackbox part, SafeTimer relies on existing protocols and their assumptions.

Abnormal events in the whitebox part may affect the processing speed of the blackbox part. SafeTimer assumes such effect can be observed at the boundary: a slow receiver may cause its NIC to drop packets because the receiver's buffer is full and SafeTimer assumes the NICs can provide packet drop statistics. We find this function is commonly provided by modern NICs.

With the help of SafeTimer, existing timeout detection protocols only need to make conservative assumptions about the blackbox part, which means the protocol can use a shorter timeout interval to accelerate failure detection. Note that SafeTimer cannot make concrete suggestions about timeout interval: the user still has to estimate possible delays in the blackbox part. However, considering the various kinds of abnormal events in the whitebox part (Section 2), SafeTimer should be able to reduce timeout interval by at least tens of seconds.

Case studies. We present a few existing timeout detection protocols to show how SafeTimer models them and how they can benefit from SafeTimer.

Budhiraja et al. [5] discuss how to detect failures in primary-backup protocols, given different models. In the simplest model, which assumes clocks are sufficiently synchronized, links are reliable, and packet delay is bounded (δ), the sender can send heartbeats every τ seconds and the receiver reports a failure if it does not receive a heartbeat for $\delta + \tau$ seconds. SafeTimer can model this protocol in the following way: when the receiver receives a heartbeat at t, it creates a new interval from t to $t + \delta + \tau$ and checks whether it receives a heartbeat by the end of the new interval; the sender can define a successful heartbeat sending for interval i as sending a heartbeat at t_i and $t_i \leq t_{i-1} + \tau$. With the help of SafeTimer, this protocol may reduce δ because it does not need to include the delays of the whitebox part. This work also discusses more complicated models, which consider link failures and proposes a gossip protocol to route heartbeats through multiple links, which is adopted in Ceph. SafeTimer can model it accordingly. For example, to tolerate one link failure, the sender can define a successful heartbeat sending as sending two heartbeats to two nodes by $t_{i-1} + \tau$. Similarly, SafeTimer may help to reduce δ.

```
/* The application calls safetimer_check when
      missing heartbeats from $start_i$ to $end_i$ */
function safetimer_check($start_i$)
    send a barrier to itself
    wait for barrier (with a timeout)
    if barrier received and $t_{lastHeartbeat}$ < $start_i$
        read drop count in OS and NIC and reset to 0
        if (drop count = 0 and $t_{drop} < start_i$)
            return TRUE_FAILURE
        else if (drop count != 0)
            $t_{drop}$ = current_time()
        end
    end
    return FALSE_FAILURE

function safetimer_recv_thread()
    when receiving heartbeat
        $t_{lastHeartbeat}$ = current_time()
    when receiving barrier
        notify safetimer_check
```

Figure 2: Pseudo code of SafeTimer's receiver module. For simplicity, it assumes there is only one sender, but it can easily be extended to support multiple senders. $t_{lastHeartbeat}$ records the timestamp of the last heartbeat. t_{drop} records the timestamp of the last drop event.

In HDFS, a DataNode sends a heartbeat to the NameNode every three seconds, and the NameNode marks the DataNode as stale if it misses heartbeats for 30 seconds. In the common case, the NameNode will acknowledge a heartbeat to the DataNode; if the DataNode detects errors, it will send heartbeats more aggressively every second. SafeTimer can model it in the following way: when the receiver receives a heartbeat at t, it creates a new interval from t to $t + 30$ and checks whether it receives a heartbeat by the end of the new interval (note intervals can overlap in this case); the sender can define a successful heartbeat sending for interval i as 1) getting acknowledgement for one heartbeat or 2) sending heartbeats with an interval of less than one second. SafeTimer may help to reduce the 30-second interval because it does not need to consider delays in the whitebox part.

4 Design

SafeTimer enhances existing timeout detection protocols to tolerate long processing delays in the whitebox part. In this section, we first present SafeTimer's mechanisms and then prove its accuracy and completeness.

4.1 Accurate timeout at the receiver

As discussed in Section 3, SafeTimer assumes the application's heartbeat receiver defines multiple time intervals (interval i from $start_i$ to end_i), and reports a failure if no heartbeat is received during an interval.

SafeTimer guarantees that as long as the receiver's NIC has processed (either delivered or dropped) a heartbeat during interval i, SafeTimer's receiver module will not report a failure for interval i.

Its key idea is simple: if the receiver module does not receive any heartbeats by the end of an interval, it will check whether there are any pending or dropped heartbeats in its whitebox part, and if not, the receiver module can safely report a failure.

The key challenge, however, is how to implement this idea in modern OS. For fast packet processing, modern OS incorporates a highly concurrent design, which involves a pipeline with multiple threads in each stage. To identify whether some heartbeats are pending, a naive solution is to pause all threads and check all buffers, but this solution will have negative impact on performance and require intrusive modification to the kernel.

To solve this problem, SafeTimer incorporates a non-blocking design as shown in Figure 2: if the application does not receive any heartbeat by end_i, it will check whether any heartbeats are pending or dropped by calling *safetimer_check*, which sends a barrier packet to itself (line 3). By crafting the barrier packet and configuring the system properly, SafeTimer ensures that a barrier will follow the same execution path of heartbeats. Therefore, if the receiver module receives the barrier, it can know that any heartbeats processed by the NIC before end_i must have been processed by the OS and SafeTimer as well, either delivered to the receiver module or dropped. We will present details about how to implement the barrier mechanism in Section 5. For now, the readers can simply assume SafeTimer somehow drives the heartbeats and the barriers into a FIFO channel.

If the receiver module receives the barrier, it will check again whether it has received a heartbeat ($t_{lastHeartbeat} < start_i$ in line 5). If not, the receiver module will read drop statistics from both the OS and the NIC: if $dropcount = 0$ and $t_{drop} < start_i$ (line 7), which means there are no drops in interval i, the receiver module can safely report a failure. If the barrier is dropped as well, the receiver module will not report a failure for interval i. In this case, the application will perform the same check in the following intervals and will eventually report a failure.

4.2 Stop sender when missing heartbeat

As discussed in Section 3, SafeTimer assumes that the application has rules to decide when to send heartbeats and whether they are sent successfully. In particular, without losing generality, SafeTimer assumes for each interval i, the application defines a deadline end'_i to send heartbeats, which should be earlier than end_i at the receiver side because of clock drift and network latency.

SafeTimer guarantees that if a sender cannot successfully send heartbeats by end'_i, the sender will not be able to send out any other packets after end'_i, because the receiver may report a failure at that time. This is necessary because the accuracy property requires that if the receiver reports a failure, the sender must have failed: violating this property can cause correctness issues. Taking the primary backup protocol as an example, a backup should only become active if the primary fails. If a backup receives a failure report and becomes active while the primary is still active, there will be two active nodes, creating a classic "split brain" problem [20].

```
1  function safetimer_send_heartbeat($end'_i$, $end'_{i+1}$)
2      send heartbeats
3      if sending succeeded before $end'_i$
4          $t_{valid} = end'_{i+1}$
5      end

7  function safetimer_intercept_sending()
8      if (current_time() > $t_{valid}$)
9          drop the packet
10     else
11         perform the send
12     end
```

Figure 3: Pseudo code of SafeTimer sender module. The application defines end'_i as the deadline to send heartbeats for interval i; the application defines whether sending succeeds; SafeTimer maintains a timestamp t_{valid} to identify till when it is safe to send out packets.

Killing a sender when it is slow is not a new idea [8, 22], but how to implement it correctly despite arbitrary processing delays requires careful thought. Existing solutions ask a specific component (e.g., a watchdog [22]) to actively kill the sender. When considering arbitrary processing delays, however, such active solution is incomplete, because the delay of processing the "kill" command may allow the sender to be alive for an arbitrary amount of time, violating the accuracy property.

SafeTimer uses a passive solution by utilizing the idea of output commit [41]: a slow sender may continue processing, but as long as other nodes do not observe the effects of such processing, the slow sender is indistinguishable from a failed sender. As shown in Figure 3 (lines 3-12), SafeTimer's sender module maintains a timestamp t_{valid}, which indicates it is safe for the sender to send packets before t_{valid}. During startup, the sender sets t_{valid} to end'_0. If the sender successfully sends heartbeats for interval i, the sender extends t_{valid} to end'_{i+1} (line 4). Whenever the sender is about to send a packet, SafeTimer will compare the current time with t_{valid}: if current time is larger than t_{valid}, the sender will discard the packet (lines 7-12). Since heartbeat is blocked as well in this case, an invalid sender cannot extend t_{valid} and send packets in the future, unless with recovery operations.

Note that since the sending operation itself may take arbitrarily long, SafeTimer allows a packet generated before t_{valid} to be actually sent out after t_{valid}. This is fine because the packet is generated when the sender is still valid (i.e., when the receiver has not reported the failure).

4.3 Proof of accuracy and completeness

As discussed in Section 3, SafeTimer relies on the existing protocol to send and receive heartbeats in the blackbox part. When the existing protocol's assumptions about the blackbox part hold, we can prove that SafeTimer is accurate (i.e., never report failure for a correct node) despite arbitrary delays in the whitebox part and is complete (i.e., eventually report failure for a failed node) when the receiver does not experience slow processing or packet drops for sufficiently long. We provide the detailed proof in the appendix.

4.4 Benefit of SafeTimer

Because of the accuracy and completeness properties, the users of SafeTimer do not need to make conservative timing assumptions about the whitebox part. They do need to provide a reasonable estimation of such delay in the common case, because the sender needs some time to send out heartbeats. However, this requirement is only for performance: if the actual delay is longer than estimation, which means the sender cannot send the heartbeat in time, SafeTimer will block the sender, which may cause unnecessary recovery and hurt performance, but this will not violate accuracy. Therefore, SafeTimer only requires the user to provide a reasonable estimation to make sure such events are *rare*. As a comparison, in existing protocols, if the actual delay is longer than estimation, system correctness can be violated, and that is why existing systems require conservative assumptions so that such events *never* happen. The gap between "rare" and "never" is where SafeTimer gains its benefit.

5 Implementation

This section presents the barrier mechanism at the receiver and the packet checking at the sender in detail.

5.1 Barrier mechanism at the receiver

The goal of the barrier mechanism is to ensure that if SafeTimer's receiver module sent a barrier to itself at t and received it later, then all heartbeats delivered by NIC before t must have been either delivered to the application or dropped. Achieving this property would be trivial if the OS processes all packets in FIFO order, but unfortunately, this is not true in modern OS. To illustrate the problem and motivate our design, we first present how Linux processes incoming packets.

Background. As shown in Figure 4, Linux incorporates a multi-stage pipeline to process incoming packets.

At the lowest level, an NIC buffers incoming packets in its RX queues and tries to transfer them to kernel's ring buffers: if the ring buffer has empty slots, the NIC will transfer the packet using DMA and fire an interrupt; if the buffer is full, the NIC will retry and may drop packets. For efficiency, modern NIC and Linux incorporate the Receive Side Scaling (RSS) technique [40] to allow parallel packet processing: the NIC creates multiple RX queues and the kernel creates an equal number of ring buffers so that each RX queue is mapped to a unique ring. Furthermore, Linux assigns a unique interrupt request (IRQ) number to each RX queue so that Linux can handle interrupts from different RX queues in parallel.

For efficiency, Linux separates interrupt handling into two parts—hard IRQ and soft IRQ—and invokes hard IRQ first. For an NIC interrupt, its hard IRQ simply sets some registers and triggers a soft IRQ. The soft IRQ reads packets from the ring buffer and executes the logic of the networking protocol, such as TCP/IP. The RSS technique allows Linux to handle IRQs in parallel.

By default, the soft IRQ reads from the ring buffer and executes the protocol logic within a single critical section protected by the lock of the ring. For more parallelism, Linux incorporates the Receive Packet Steering (RPS) technique [40]: when RPS is enabled, a soft IRQ reads a packet from the ring, puts it into a buffer called *backlog*, and then releases the lock of the ring. A separate thread, which may run on another CPU, will retrieve packets from the backlog and execute the protocol logic.

Finally the soft IRQ puts packets into socket buffers and the user-space threads may read from these buffers in parallel.

Such a multi-stage pipeline may re-order packets. Modern NIC and Linux preserve FIFO order for TCP packets with the same (sender IP, sender port, destination IP, destination port) and UDP packets with the same (sender IP, destination IP), by directing packets with same such information to the same RX queue, backlog and socket buffer. For SafeTimer, such guarantee is not enough since heartbeats and barriers are from different senders.

Overview of SafeTimer's solution. Our implementation is driven by three principles: 1) for portability, we hope to minimize modification to OS kernel code; 2) for performance, it should not incur significant overhead; 3) for portability, we hope to minimize dependence on specific NIC features or modification to NIC drivers.

As shown in Figure 4, SafeTimer re-directs heartbeats and barriers to a separate FIFO queue (called STQueue) early in the pipeline, so that they are not affected by re-ordering in later stages. However, since the earliest place we can perform such re-direction is after the soft IRQ reads the packets, RSS technique in the earlier stage may still re-order packets from different ring buffers. To solve this problem, SafeTimer sends a barrier packet to each RX queue/ring. If all of them later go through the

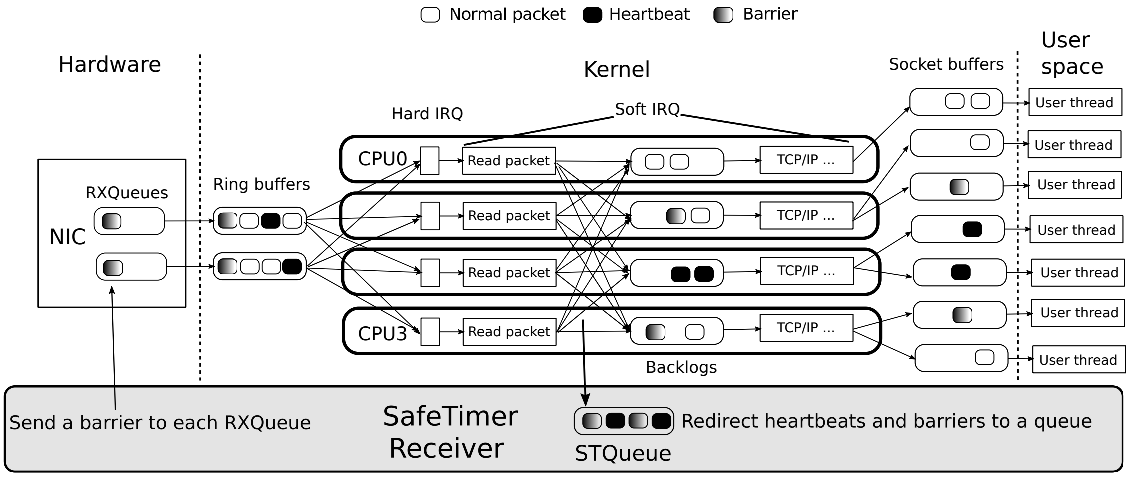


Figure 4: Barrier mechanism at the receiver. The algorithm in Figure 2 reads from STQueue.

STQueue, SafeTimer can know that all previous heartbeats are processed. The key to the correctness of this approach is that a soft IRQ needs to grab the lock of the ring buffer when reading a packet from the ring, and thus packets from each ring are read in a FIFO manner. As long as SafeTimer re-directs a packet before the soft IRQ releases the lock, such per-ring FIFO order will be retained in the STQueue. Therefore, when SafeTimer retrieves a barrier from the STQueue, it knows all previous heartbeats from the same ring must have been processed.

Next we present each step in detail.

Forcing a barrier to go through NIC. SafeTimer requires a barrier packet to follow the same execution path of a heartbeat packet. Putting a barrier in the ring buffer does not work because the OS won't read from the buffer until an NIC interrupt is triggered. Therefore, SafeTimer receiver forces the barrier packet to go through its NIC. This task, however, is challenging for multiple reasons.

First, Linux has the loopback optimization to route a local packet by memory copy instead of sending it to the NIC. SafeTimer bypasses this optimization by sending the barrier directly to the device driver. This approach, however, creates a new problem: the NIC will actually send the packet to the router. To prevent loops, routing protocols usually have a constraint that a router should never forward a packet to the port where the packet is received. Therefore, the router will drop a barrier packet, whose destination and source are the same.

Our prototype uses an NIC with two ports and sends a barrier from one port to the other, which eliminates the above problem. This solution requires the receiver to have at least two links to the router, but considering the fact that redundant links are already widely used for fault tolerance, such requirement often does not incur additional cost. If redundant link is not available, another alternative is to use the virtual LAN (vLAN) technique to virtualize a physical port into two virtual ports [47].

Sending a barrier to a specific RX queue. A few NICs provide the "N-tuple filter" feature to direct packets to specified RX queues, which makes this problem trivial. However, we find this feature is not common so far [21]. Most NICs calculate a hash value based on the IPs and ports information in a packet and then direct the packet to an RX queue based on the hash value. Therefore, we propose a general solution based on the assumption that one cannot control which RX queue a packet is directed to, but packets with same IPs and ports will always be directed to the same RX queue.

SafeTimer uses a brute-force search approach: during initialization, its receiver module sends barriers with different sender ports to its NIC to see which RX queue they are directed to, until SafeTimer can find a port for each RX queue. Since usually there are not many RX queues, such procedure could finish quickly. The challenge, however, is how to know which RX queue (represented by its IRQ number) a packet is directed to. SafeTimer uses netfilter [39], which is a tool provided by Linux, to intercept soft IRQ functions to check whether a packet is a barrier, but soft IRQ functions do not carry the IRQ number of the RX queue. We can modify the driver to pass the IRQ number to the soft IRQ, but this violates our principle to minimize driver-specific modifications.

To solve this problem, we leverage the *irq-cpu affinity* configuration provided by Linux, which can configure the mapping between RX queues and CPUs during RSS. By default, it is configured to be an all-to-all mapping, which means any CPU can execute any IRQ to read from its corresponding RX queue/ring, but Linux also allows one-to-one mapping. We leverage this option to "test" whether a barrier is sent to a specific IRQ *i*: we map IRQ *i* to CPU 0 and the other IRQs to the remaining CPUs arbitrarily. When intercepting the soft IRQ function, Safe-

Timer reads the CPU ID: if the packet is a barrier and the IRQ function is run on CPU 0, we can know the barrier must be sent to IRQ *i*; otherwise, SafeTimer tests a different *i* until it can find the right one.

Note that since the NIC always directs packets with same IPs and ports to the same RX queue, we only need to run the inferring procedure once for one machine. Afterwards we can use all-to-all mapping for efficiency.

Re-directing packets to STQueue. As shown in Figure 4, SafeTimer re-directs heartbeats and barriers to a FIFO STQueue after packets are read.

To implement this functionality, SafeTimer uses *netfilter* to hook the *ip_local_deliver* function, and configures iptable to re-direct heartbeats and barriers to a FIFO netfilter queue, which is called STQueue in SafeTimer. SafeTimer hooks *ip_local_deliver* because this is the earliest point packets can be re-directed in *netfilter*. SafeTimer sends heartbeats and barriers to specific ports so that they can be efficiently distinguished from normal packets.

This approach, however, is not fully correct when RPS is enabled: recall that when RPS is enabled, a soft IRQ will put a packet into the backlog and then releases the lock of the ring. In this case, *ip_local_deliver* is called after the lock is released and thus re-direction may not preserve the order of packets from the corresponding ring. To solve this problem, we use kretprobe [33] to intercept *get_rps_cpu* to return -1 for heartbeats and barriers: doing so essentially disables RPS for heartbeats and barriers. As a result, the re-direction will be executed under the protection of the lock of each ring and thus STQueue will preserve the order of packets from each ring. Normal packets, however, are not affected.

The timeout detection protocol (Figure 2) always reads heartbeats and barriers from the STQueue. However, SafeTimer does not remove heartbeats and barriers from later stages of the pipeline, because the OS needs to execute the logic of the network protocol, like congestion control or sending acknowledgements in TCP.

Reading drop count. SafeTimer's receiver module needs to read packet drop counts from both the OS and the NIC. Linux and most NICs have provided such statistics, but their implementation cannot achieve our goal.

In Linux, the NIC device driver periodically reads the drop count from the NIC, which can be fetched by reading /proc files system or using tools such as ethtool. Periodic reading means such statistics may be stale, which can cause SafeTimer's receiver module to miss recent drops and generate a false failure report. To make things worse, the NIC will reset drop count to 0 after it is read, so even if SafeTimer reads the drop count directly from the NIC, it may still get inaccurate results. To solve this problem, SafeTimer reads drop count from the NIC and then merges it with the number reported by the NIC driver. This is the only place SafeTimer requires modification to device drivers and OS kernel.

5.2 Blocking slow sender

As shown in Figure 3, SafeTimer's sender module blocks the sender if it cannot deliver heartbeats to the NIC in time. However, when sending a packet, Linux does not notify users whether or not the packet is delivered to the NIC successfully. Instead, it may write the packet to a buffer, return to the user, and send the packet to the NIC later, which may fail. To solve this problem, we use *kprobe* to intercept the function that the NIC driver invokes to reclaim resources after transmission is complete (e.g., *napi_consume_skb* or *__dev_kfree_skb_any*). As shown in Figure 3, SafeTimer applies the rules of the existing timeout detection protocol to check whether heartbeats are sent successfully. If so, SafeTimer's sender module will update t_{valid}. To block invalid packets, we use *netfilter* to intercept the *ip_output* function: if current time is larger than t_{valid}, the packet will be dropped.

Because of the processing delay, SafeTimer cannot get the exact time when a packet is sent. Instead, SafeTimer conservatively uses the timestamp after sending a packet, t_{after}: when checking whether a heartbeat is sent before end_i^{l} (line 3 in Figure 3), SafeTimer compares t_{after} with end_i^{l}. Such conservative approach ensures a sender failing to send heartbeats in time must be blocked, but it may also block a sender that has sent heartbeats in time, which is unnecessary but does not violate accuracy. Previous works have discussed how to minimize the impact of such unnecessary killing [38].

Since a slow sender process may communicate with other processes on the same machine, SafeTimer needs to block those processes as well, and thus it provides two blocking modes: the first blocks all processes on a machine; the second blocks only the sender process if the user is sure it does not communicate with other processes. Automatically tracking the information flow among different processes is out of the scope of this paper.

5.3 Supporting virtual machine

To maximize the benefit of SafeTimer in a virtual machine architecture, we could implement SafeTimer in the host OS or hypervisor and provide related functions to applications using hypercalls or remote procedure calls. By doing so, we can model the host OS or hypervisor as a whitebox. We plan to implement such support in the future. However, if the user has no control of the host OS or hypervisor, he/she can still deploy SafeTimer to the guest OS and model the host OS/hypervisor as a blackbox, but this approach of course loses the ability to tolerate long delays in the host OS/hypervisor.

6 Evaluation

Our evaluation tries to answer three questions:

- What is the overhead of SafeTimer?
- Can SafeTimer achieve the expected accuracy property, despite long delays in the OS and the application?
- How much effort does it take to apply SafeTimer to existing systems?

To answer the first question, we have evaluated SafeTimer with a performance benchmark, which can send packets with different sizes, and compared its throughput and latency to those without SafeTimer. For the blackbox part, we use a simple protocol that sends heartbeats periodically with a configurable interval.

To answer the second question, we have injected long delays and packet drops at different layers at both the sender and the receiver to observe whether SafeTimer can prevent false failure report. Of course, this is by no means a complete test: we have proved the accuracy of SafeTimer in the appendix. This set of experiments serves as a sanity check about whether our implementation has actually achieved the expected properties.

To answer the third question, we have applied SafeTimer to HDFS and Ceph to enhance their timeout detection protocols and report our experience.

Testbed setting. We ran all experiments on CloudLab [15]. Each machine is equipped with two Intel Xeon E5-2630 8-core CPUs, 128GB of memory, 1.2 TB of SAS HDD, and a dual-port Intel X520-DA2 10Gb NIC. All machines are connected to a Cisco Nexus C3172PQs switch. Linux 4.4.0 is installed on all machines.

6.1 Overhead

SafeTimer incurs overhead for each packet at both the sender and the receiver: SafeTimer's sender module compares current time with t_{valid} before sending each packet; SafeTimer's receiver module re-directs heartbeats and barriers to the STQueue. To know whether a packet is a heartbeat or a barrier, the receiver module checks the destination port of each packet. When a sender fails, SafeTimer performs additional operations to block the sender, send barriers, and read drop counts, but since failure is rare, we focus on overhead in the failure-free case.

Since SafeTimer incurs overhead for each packet, such overhead should be relatively higher for workloads with smaller packets and thus we measure the overhead of SafeTimer with different packet sizes. However, TCP may merge small packets in the same connection and thus affect our experiment results. To prevent such effect, we use a ping-pong benchmark as suggested in a previous work [2]: we create multiple sender threads at the sender, each creating a connection to the receiver. The sender thread sends a packet to the receiver and waits for the receiver to forward the packet back. In this case, since each connection has only one outstanding packet, TCP has no chance to merge packets. To increase load, we can increase the number of sender threads.

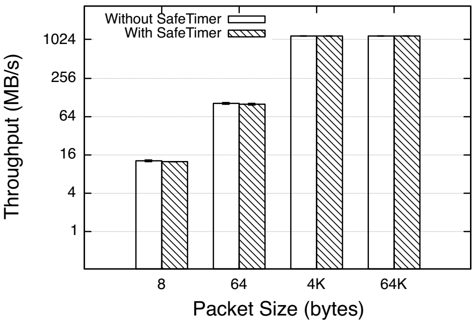


Figure 5: Throughput of the ping-pong benchmark with and without SafeTimer.

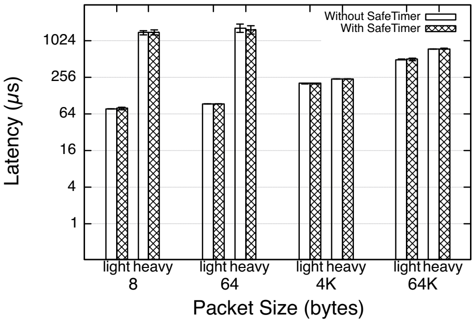


Figure 6: 99 percentile latency of the ping-pong benchmark with and without SafeTimer.

To measure the overhead of SafeTimer, we apply SafeTimer to the ping-pong benchmark and measure how it affects throughput and latency. To measure the maximal throughput, we increase the number of sender threads till we cannot gain higher throughput. To measure the latency, we run experiments under two loads: a light load of about 40% of the maximal throughput and a heavy load of about 90% of the maximal throughput. We do not measure the latency under the maximal throughput because in this case, the latency will be dominated by queuing delay. We run each setting 20 times to compute the average and standard deviation. We set the timeout interval of the blackbox part to be one second.

As shown in Figures 5 and 6, SafeTimer's overhead is small: for 4KB and 64KB packets, the overhead is less than 1%; for 8B and 64B packets, SafeTimer can

Node	Instrument Position	Injected Event	SafeTimer	Vanilla
Receiver	System call (recv)	Delay	No timeout	Timeout
Receiver	Socket (sock_queue_rcv_skb)	Delay/Drop	No timeout	Timeout
Receiver	NFQueue (nfqnl_enqueue_packet)	Delay/Drop	No timeout	N/A
Receiver	IP (ip_rcv)	Delay	No timeout	Timeout
Receiver	RPS (enqueue_to_backlog)	Delay/Drop	No timeout	Timeout
Receiver	Ethernet (napi_gro_receive)	Delay	No timeout	Timeout
Sender	System call (send)	Delay	Blocked	Alive
Sender	Socket (sock_sendmsg)	Delay	Blocked	Alive
Sender	IP (ip_output)	Delay/Drop	Blocked	Alive. Can observe drop.
Sender	Ethernet (dev_queue_xmit)	Delay	Blocked	Alive

Table 1: Verifying accuracy of SafeTimer by injecting long delay or packet drops. Gray cells indicate injection in kernel. N/A means this test case does not apply.

increase 99p latency by 0.7% to 2.7% and decrease throughput by 1.6% to 2.4%. Such low overhead is reasonable because SafeTimer's additional work (i.e., comparing t_{valid} at the sender and reading destination port at the receiver) is small compared to other work the OS has to perform for each packet (e.g., interrupt handling, memory copy). To confirm the result, we run the same benchmark on another set of machines on CloudLab (m510 [16]) with different NICs (Mellanox ConnectX-3 10G) and we find the overhead of SafeTimer is similar.

6.2 Accuracy

Although we have proved the accuracy of SafeTimer, we hope to sanity check whether our implementation has achieved the expected property. For this purpose, we inject long delays and packet drops at different layers at the sender and the receiver. We compare SafeTimer to a vanilla timeout implementation, which has a user thread to periodically send heartbeats at the sender and a user thread to periodically check timeout at the receiver.

Table 1 summarizes the events we injected and how SafeTimer responds to these events. We inject long delays at all positions but only inject drops if the corresponding function can actually drop packets. In these experiments, we set timeout interval to be one second and inject a delay of two seconds. As shown in the table, SafeTimer correctly prevents false failure report at the receiver and blocks the sender in all cases. The vanilla implementation, however, violates accuracy in almost all cases except when a heartbeat is dropped in *ip_output*: in this case, the sender receives an error and can retry.

6.3 Case studies

To evaluate how much effort it takes to apply SafeTimer to real-world applications and its performance overhead, we have applied SafeTimer to HDFS [45] and Ceph [9].

APIs of SafeTimer. At the sender side, SafeTimer provides two APIs: *safetimer_send_HB* to send a heartbeat and check whether it is delivered to the NIC in time; *safetimer_extend* to extend the *t_valid* value. At the receiver side, SafeTimer provides one API: *safetimer_check* to check whether it is safe to report a failure.

HDFS. In HDFS, a DataNode needs to periodically send a heartbeat to the NameNode and if the NameNode misses a number of consecutive heartbeats, the NameNode will mark the DataNode as "stale".

We modified one line of code in NameNode's *isStale* function, which checks whether heartbeats are missing for a DataNode, to perform the additional *safetimer_check*. We modified six lines of code in DataNode to use SafeTimer's APIs to send heartbeats and check whether heartbeats are sent in time. To simplify modification, we do not remove HDFS' original heartbeat mechanism: this leads to duplicate heartbeats but during our experiments, the overhead is negligible.

We killed a DataNode and found the NameNode can correctly mark a failed DataNode as stale. We have measured the performance of an HDFS deployment with three DataNodes by using Hadoop's built-in benchmark tool DFSIO. We ran each experiment five times. Without SafeTimer, DFSIO can achieve a write throughput of 203 MB/s (stdev 12.6) and a read throughput of 627 MB/s (stdev 18.4); with SafeTimer, it can achieve a write throughput of 206 MB/s (stdev 5.5) and a read throughput of 632 MB/s (stdev 8.4). The difference is not statistically significant.

Ceph. In Ceph, an Object Storage Daemon (OSD) sends heartbeats to its two peers every 6 seconds and if they can't receive the heartbeat for 20 seconds, they will send a failure report to the Monitor, which will consider the OSD as failed if receiving two reports.

In this mechanism, an OSD is both the sender and receiver of heartbeats. We modified two lines of code in OSD's *heartbeat_check* function to perform the *safetimer_check* before sending the failure report; we modified five lines of code to use SafeTimer's APIs to send heartbeats and check whether heartbeats are sent in time.

We killed an OSD and found the Monitor can mark it as down. We have measured the performance of a Ceph deployment with three OSDs by using Ceph's inbuit benchmark tool RADOS. We ran each experiment five times. Without SafeTimer, RADOS can achieve a bandwidth of 43.3 MB/s (stdev 1.6); with SafeTimer, it can achieve a bandwidth of 42.2 MB/s (stdev 1.1). The difference is not statistically significant.

7 Related work

Chandra et al. show that many classic problems in distributed system, such as consensus, can be solved with an accurate and complete failure detector [12]. In practice, timeout is widely used for failure detection, whose accuracy depends on their timing assumptions.

Synchronous systems. Under synchronous assumptions (i.e., delay of message transfer and clock deviation are bounded [12]), timeout can achieve both accuracy and completeness for failure detection. Many systems like primary-backup replication and HDFS [4, 5, 18, 24, 45] work under this assumption. To guarantee accuracy, these systems must make conservative assumptions about message delay and clock deviation. Previous works have tried to improve its accuracy by estimating the upper bound adaptively at runtime [3] and by killing a node if the failure detector reports the node has failed [8, 22]. SafeTimer can enhance synchronous systems to tolerate abnormal events in the OS and the application, without requiring any timing assumptions.

Asynchronous systems. Under asynchronous assumptions (i.e., delay of message transfer and clock deviation are unbounded), building a failure detector that is both accurate and complete is proved to be impossible [23]. Paxos [34, 35, 42] is a replication protocol designed for asynchronous environments: it is always correct (i.e., all correct replicas process the same sequence of requests) and is live (i.e., the system can make progress) when the environment is synchronous for sufficiently long. Paxos is used as building blocks in larger systems like Spanner [17] and Microsoft Azure Storage [7]. Compared to synchronous replication systems, Paxos is more expensive in terms of number of replicas and messages. Asynchronous systems don't need accurate failure detection for correctness, but since there is a cost to recover a failure, SafeTimer may help to reduce such unnecessary recovery by reducing the number of false failure reports.

Lease systems. A number of systems [1, 13] install a replicated lease manager (e.g., Chubby [6] and ZooKeeper [29]): a server needs to acquire a lease from the lease manager before it can service clients; the server has to renew the lease before it expires, and if not successful, the server will stop servicing clients. For accuracy, this approach requires the clock speed of servers and the lease manager to be sufficiently close, but it does not require the delay of message transfer to be bounded. Lease systems strike a balance between cost and timing assumptions, but it has its own limitations: first, the centralized nature of the lease manager means if a long delay happens at the lease manager, all leases will expire and all servers will stop servicing, which does not violate the accuracy property, but is certainly undesirable. As a result, lease systems prefer coarse-grained leases [6], which hurts system availability as well, similar as using a long timeout. Second, the requirement of a replicated lease manager makes it less desirable in small-scale systems. Systems using leases can benefit from SafeTimer by installing its sender module to ensure a server will not continue servicing after its lease expires.

Failure detection without timeout. A few systems implement a failure detector without using timeout. For example, Falcon [38] and its following works [36, 37] install probes in routers to monitor servers and install probes at different layers in a server to monitor upper layers. This approach essentially converts the whole communication channel into a white box. As a result, it requires intrusive modification to the routing layer, which makes its deployment challenging and sometimes impossible if the routers are out of the control of the user. To solve these problems, Falcon uses timeout as a backup.

Real-time OS. Real-time Linux [43] and other real-time frameworks for Linux such as RTAI [44] and Xenomai [49] can guarantee important tasks or interrupts are scheduled before given deadlines. However, this is not sufficient to achieve our goal, because long delay is not only caused by untimely scheduling, but also caused by the fact that an important task is occasionally blocked by a heavy task (Section 2). Real-time scheduling can address the former problem, but not the latter one.

8 Conclusion

This paper shows that we do not need to include the maximal local processing delay in timeout interval. Because of the whitebox nature of local processing, we can build efficient and accurate failure detection for this part, despite arbitrary processing delays. Our prototype SafeTimer allows one to use a shorter timeout to improve system availability, without sacrificing accuracy.

Acknowledgements

Many thanks to our shepherd Irene Zhang and to the anonymous reviewers for their insightful comments. This material is based in part upon work supported by NSF grant CNS-1566403.

References

[1] Apache HBASE. http://hbase.apache.org/.

[2] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, CO, 2014. USENIX Association.

[3] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 354–363, Washington, DC, USA, 2002. IEEE Computer Society.

[4] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault Tolerance. *ACM Transanctions on Computer Systems*, 14(1):80–107, February 1996.

[5] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations. In *CDCCA*, 1992.

[6] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[7] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[8] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[9] Ceph. https://ceph.com.

[10] Ceph Default Heartbeat Configuration. http://docs.ceph.com/docs/master/rados/configuration/mon-osd-interaction/.

[11] Ceph MDS heartbeat timeout during rejoin. http://tracker.ceph.com/issues/19118.

[12] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[14] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[15] CloudLab. https://www.cloudlab.us.

[16] Hardware information in the CloudLab manual. http://docs.cloudlab.us/hardware.html.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[18] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, 2008.

[19] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. Understanding real-world timeout problems in cloud server systems. In *IC2E 18*.

[20] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: A survey. *ACM Comput. Surv.*, 17(3):341–370, September 1985.

[21] Overview of Networking Drivers. http://dpdk.org/doc/guides/nics/overview.html.

[22] Christof Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput.*, 52(2):99–112, February 2003.

[23] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, 2003.

[25] Hadoop Default Configuration. `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/core-default.xml`.

[26] Set the zk default timeout to 3 minutes. `https://issues.apache.org/jira/browse/HBASE-3273`.

[27] Hdfs. `http://hadoop.apache.org/`.

[28] HDFS Default Configuration. `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml`.

[29] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*, 2010.

[30] Internet control message protocol. `https://tools.ietf.org/html/rfc792`, 1981.

[31] InfiniBand Performance. `http://www.mellanox.com/page/performance_infiniband`.

[32] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T. Kandemir. Hios: A host interface i/o scheduler for solid state disks. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 289–300, Piscataway, NJ, USA, 2014. IEEE Press.

[33] Kernel Probes (Kprobes). `https://www.kernel.org/doc/Documentation/kprobes.txt`.

[34] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[35] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, December 2001.

[36] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 427–441, Lombard, IL, 2013. USENIX.

[37] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 9:1–9:16, New York, NY, USA, 2015. ACM.

[38] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *SOSP*, 2011.

[39] Netfilter. `http://www.netfilter.org/`.

[40] Scaling in the Linux Networking Stack. `https://www.kernel.org/doc/Documentation/networking/scaling.txt`.

[41] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proc. 7th OSDI*, November 2006.

[42] B. Oki and B. Liskov. Viewstamped replication: A general primary copy method to support highly-available distributed systems. In *Proc. 7th PODC*, 1988.

[43] Linux Foundation Real-Time Linux Project. `https://rt.wiki.kernel.org`.

[44] RTAI - the RealTime Application Interface for Linux. `https://www.rtai.org`.

[45] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[46] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Holzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in googles datacenter network. In *Sigcomm '15*, 2015.

[47] Patricia Thaler, Norman Finn, Don Fedyk, Glenn Parsons, and Eric Gray. Media access control bridges and virtual bridged local area networks. Technical report, IETF, March 2013.

[48] Yang Wang, Manos Kapritsos, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems. In *NSDI*, 2014.

[49] Xenomai - Real-time framework for Linux. `http://xenomai.org`.

A Appendix: Proof of accuracy and completeness

Assumption of the blackbox part. *The existing protocol can guarantee that if a sender has successfully sent heartbeats for interval i, at least one of the heartbeats will be processed (either delivered to the OS or dropped) by the receiver's NIC by end_i.*

Theorem A.1. *(Accuracy) If SafeTimer's receiver module reports a failure at time t, the sender will not be able to send any packets generated after t.*

Proof. As shown in the protocol, SafeTimer's receiver module reports a failure for interval i if two conditions are both satisfied: first, the receiver module has received the barrier but not any heartbeats for interval i. Because the barrier is sent after end_i and because of the barrier's semantic, any heartbeats processed by the NIC before end_i must either have been delivered to the receiver module or have been dropped. Since the receiver module has not received any heartbeats, we can conclude that it is either because the NIC has not processed any heartbeat by end_i or because some heartbeats are dropped at the receiver side.

The second condition is $dropcount = 0$ and $t_{drop} < start_i$, which means there are no packet drops at the receiver side in interval i. By combining this condition with the first one, we can conclude that the receiver's NIC must have not processed any heartbeat packets for interval i before end_i. This means the sender must have not successfully sent the heartbeats for interval i (Assumption of the blackbox). In this case, the sender will not extend its $t_{valid} = end_i'$, and thus will stop sending any messages after t_{valid}.

Since t is larger than end_i and $t_{valid} = end_i'$ at the sender should be earlier than end_i at the receiver, we can conclude that $t_{valid} < end_i < t$ and thus the sender will not send any packets generated after t. □

Theorem A.2. *(Completeness) If the sender has failed, SafeTimer's receiver module will eventually report a failure if the following two conditions both hold for sufficiently long (five consecutive intervals in the worst case): 1) the receiver's processing speed is normal, which means events (e.g., heartbeat, barrier, and reading drop count) generated before or during an interval can be handled by the end of the interval; 2) the receiver does not experince any packet drops.*

Proof. Suppose the sender fails to send heartbeats in interval i, and afterwards, there are five consecutive intervals j to $j+4$ ($j > i$) during which the receiver's processing speed is normal and the receiver does not experience any packet drops.

Since the receiver's processing speed is normal in interval j, the receiver should be able to handle all delayed heartbeats from the sender, if any, by the end of interval j, which means the receiver won't receive any heartbeats in interval $j+1$. Therefore, the receiver will send a barrier at the end of interval $j+1$. Since the receiver's processing speed is normal and there are no packet drops in interval $j+2$, the receiver will receive the barrier and read drop count by the end of interval $j+2$. If drop count is 0 (t_{drop} must be smaller than $start_{j+2}$ because the receiver does not read drop count in interval $j+1$), the receiver will report the failure; otherwise, the receiver will update t_{drop} (the new t_{drop} must be smaller than $start_{j+3}$) and repeat the above procedure. At the end of interval $j+3$, the receiver must report a failure because both conditions to report a failure can be met: 1) since the processing speed is normal and there are no packet drops in interval $j+4$, the receiver can receive the barrier for $j+3$ but it cannot receive any heartbeats; 2) drop count is 0 because there are no packet drops in interval $j+3$ and $j+4$; $t_{drop} < start_{j+3}$. □

Note that five intervals are the worst case: if previously there are no delayed heartbeats or packet drops, the receiver will report the failure after one interval.

Improving Service Availability of Cloud Systems by Predicting Disk Error

Yong Xu[1], Kaixin Sui[1], Randolph Yao[2], Hongyu Zhang[3], Qingwei Lin[1],
Yingnong Dang[2], Peng Li[4], Keceng Jiang[1], Wenchi Zhang[1], Jian-Guang Lou[1],
Murali Chintalapati[2], Dongmei Zhang[1]
[1]*Microsoft Research, China* [2]*Microsoft Azure, USA*
[3]*The University of Newcastle, Australia* [4]*Nankai University, China*

Abstract

High service availability is crucial for cloud systems. A typical cloud system uses a large number of physical hard disk drives. Disk errors are one of the most important reasons that lead to service unavailability. Disk error (such as sector error and latency error) can be seen as a form of gray failure, which are fairly subtle failures that are hard to be detected, even when applications are afflicted by them. In this paper, we propose to predict disk errors proactively before they cause more severe damage to the cloud system. The ability to predict faulty disks enables the live migration of existing virtual machines and allocation of new virtual machines to the healthy disks, therefore improving service availability. To build an accurate online prediction model, we utilize both disk-level sensor (SMART) data as well as system-level signals. We develop a cost-sensitive ranking-based machine learning model that can learn the characteristics of faulty disks in the past and rank the disks based on their error-proneness in the near future. We evaluate our approach using real-world data collected from a production cloud system. The results confirm that the proposed approach is effective and outperforms related methods. Furthermore, we have successfully applied the proposed approach to improve service availability of Microsoft Azure.

1 Introduction

In recent years, software applications are increasingly deployed as online services on cloud computing platforms, such as Microsoft Azure, Google Cloud, and Amazon AWS. As cloud service could be used by millions of users around the world on a 24/7 basis, high availability has become essential to the cloud-based services. Although many cloud service providers target at a high service availability (such as 99.999%), in reality, service could still fail and cause great user dissatisfaction and revenue loss. For example, according to a study conducted on 63 data center organizations in the U.S, the average cost of downtime has steadily increased from $505,502 in 2010 to $740,357 in 2016 (or a 38 percent net change) [33].

Various software, hardware, or network related problems may occur in a cloud system. Our experience with Microsoft Azure shows that disk problem is the most severe one among hardware issues. A typical cloud system like Azure uses hundreds of millions of hard disk drives. Disk-related problem has become one of the most significant factors that contribute to the service downtime. The importance of disk problem is also observed by researchers in Facebook and Google, who reported that 20-57% of disks experience at least one sector error in datasets collected over 4-6 years [27, 35].

To improve service availability, many proactive disk failure prediction approaches have been proposed [18, 31, 32, 42, 41]. These approaches train a prediction model from historical disk failure data, and use the trained model to predict if a disk will fail (i.e., whether a disk will be operational or not) in near future. Proactive actions, such as replacement of failure-prone disks, can then be taken. The prediction model is mainly built using the SMART [1] data, which is disk-level sensor data provided by firmware embedded in disk drives.

The existing approaches focus on predicting *complete disk failure* (i.e., disk operational/not operational). However, in a cloud environment, before complete disk failure, upper-layer services could already be affected by *disk errors* (such as latency errors, timeout errors, and sector errors). The symptoms include file operation errors, VM not responding to communication requests, etc. Disk errors can be seen as a form of *gray failure* [22], which are fairly subtle failures that can defy quick and definitive detection by a conventional system failure detector, even when applications are afflicted by them. Gunawi et al. also pointed out the impact of fail-slow hardware that is still functional but in a degraded mode [20].

If no actions are taken, more severe problems or even service interruptions may occur. Therefore, we advocate that it is important to predict disk errors so that proactive measures can be taken before more severe damage to the service systems incur. The proactive measures include error-aware VM allocation (allocating VMs to healthier disks), live VM migration (moving a VM from a faulty disk to a health one), etc. In this way, service availability can be improved by predicting disk errors.

In this paper, we develop an online prediction algorithm for predicting disk errors, aiming at improving service availability of a cloud service system. We find that disk errors can be often reflected by system-level signals such as OS events. Our approach, called CDEF (stands for Cloud Disk Error Forecasting), incorporates both SMART data and system-level signals. It utilizes machine learning algorithms to train a prediction model using historical data, and then use the built model to predict the faulty disks. We design the prediction model to have the following abilities:

- Be able to rank all disks according to the degree of error-proneness so that the service systems can allocate a VM to a much healthier one.

- Be able to identify a set of faulty disks from which the hosted VMs should be live migrated out, under the constrains of cost and capacity.

However, it is challenging to develop an accurate disk error prediction model for a production cloud system. We have identified the following challenges:

1. In real-world cloud service systems, the extremely imbalanced data make prediction much more difficult. In average, only about 300 out 1,000,000 disks could become faulty every day. We need to identify the faulty disks and be careful not to predict a healthy disk as faulty. In our work, we propose a cost-sensitive ranking model to address this challenge. We rank the disks according to their error-proneness, and identify the faulty ones by minimizing the total cost. Using the cost-sensitive ranking model, we only focus on identifying the top *r* most error-prone disks, instead of classifying all faulty disks. In this way, we mitigate the extreme imbalance problem.

2. Some features, especially system-level signals, are time-sensitive (their values keep changing drastically over time) or environment-sensitive (their data distribution would significantly change due to the ever-changing cloud environment). We have found that models built using these unstable features may lead to good results in cross-validation (randomly dividing data into training and testing sets) but perform poorly in real-world online prediction (dividing data into training and testing sets by time). We will elaborate this challenge in Section 2.2. To address this challenge, we perform systematic feature engineering and propose a novel feature selection method for selecting stable and predictive features.

We evaluate our approach using real-world data collected from a production cloud system in Microsoft. The results show that CDEF is effective in predicting disk errors and outperforms the baseline methods. We have also successfully applied CDEF in industrial practice. In average, we successfully reduce around 63k minutes of VM downtime of Microsoft Azure per month.

In summary, we make the following contributions in this paper:

- We propose CDEF, a disk error prediction method. In CDEF, we consider both system-level signals and disk-level SMART attributes. We also design a novel feature selection model for selecting predictive features and a cost-sensitive ranking model for ranking disks according to their error-proneness.

- We have successfully applied CDEF to Azure, a production cloud system in Microsoft. The results prove the effectiveness of CDEF in improving service availability in industrial practice. We also share the lessons learned from our industrial practice.

The rest of this paper is organized as follows: In Section 2, we introduce the background and motivation of our work. Section 3 describes the proposed approach and detailed algorithms. The evaluation of our approach is described in Section 4. We also discuss the results and present the threats to validity. In Section 5, we share our experience obtained from industrial practice. The related work and conclusion are presented in Section 6 and Section 7, respectively.

2 Background and Motivation

2.1 Disk Error Prediction

A cloud system such as Microsoft Azure contains hundreds of millions of disks serving various kinds of services and applications. Disks are mainly used in two kinds of clusters, clusters for data storage and clusters for cloud applications. For the former of clusters, redundancy mechanisms such as RAID [30] could tolerate disk failures well. The latter form of clusters hosts a tremendous amount of virtual machines, disk errors could bring undesirable disruptions to the services and applications. In this paper, we focus on the disks used in the cloud application cluster.

For cloud systems such as Microsoft Azure, Amazon AWS, and Google Cloud, service problems can lead to great revenue loss and user dissatisfaction. Hence, in today's practice, the service providers have made every effort to improve service availability. For example, from "four nines" (99.99%) to "five nines" (99.999%), and then to "six niness"(99.9999%). Disks are among the most frequently failing components in a cloud environment and have attracted much attentions from both academia and industry. For example, BackBlaze publishes quarterly reports and the underlying data for users to keep track of reliability of popular hard drives in the market. In their data, disk failure is labelled *0 if the drive is OK, and 1 if this is the last day the drive was operational before failing* [2].

To mitigate cost incurred by disk failures, researchers have proposed to automatically predict the occurrence of disk failure before it actually happens. In this way, proactive measures, such as disk replacement, can be taken. Disk failure prediction has been a hot subject of study. Existing work [9, 18, 31, 32, 41, 42] mostly use the SMART data (Self-Monitoring, Analysis and Reporting Technology, which monitors internal attributes of individual disks) to build a disk failure prediction model.

However, before a disk completely fails, it already started reporting errors. There are various disk errors such as disk partition errors (disk volumes and volume size become abnormal), latency errors (unexpected long delay between a request for data and the return of the data), timeout errors (exceeding the predefined disk timeout value), and sector errors (individual sectors on a drive become unavailable), etc. Disk failures can be detected by a conventional system failure detection mechanisms. These mechanisms often assume an overly simple failure model in which a component is either correct or failed. However, such mechanisms are inadequate to deal with disk errors as they are subtle *gray failures* [22]. In our practice, the disk error data is obtained through root cause analysis of service issues performed by field engineers.

Disk errors are common. For example, a study by Bairavasundaram et al. [8] reports that 5-20% of hard disk drives in Netapps storage systems report sector errors over a period of 24 months. The disk errors can affect the normal operations of upper-layer applications and can be captured by unexpected VM downtime. The symptoms include I/O requests timeout, VM or container not responding to communication requests, etc. If no actions are taken, more severe problems or even service interruptions may occur. Therefore, it is important that disk errors to be captured and predicted before the virtual machines get affected.

2.2 Challenges

In this work, we propose to predict the error-proneness of a disk based on the analysis of historical data. The ability to predict disk error can help improve service availability from the following two aspects:

- VM allocation, which is the process of allocating a VM (virtual machine) to a host. To enable more effective VM allocation, we can proactively allocate VMs to a host with a healthier disk rather than to a host with a faulty disk.
- Live migration, which is the process of moving a running VM among hosts without disconnecting the client or application. To enable more effective live migration, we can proactively migrate VMs from a host with a faulty disk to a host with healthy disks.

To achieve so, we can build a prediction model based on historical disk error data using machine learning techniques, and then use the model to predict the likelihood of a disk having errors in the near future. There are several main technical challenges in designing the disk error prediction model for a large-scale cloud:

Extremely imbalanced data: For a large-scale cloud service system such as Microsoft Azure, each day, at most only 3 disk in ten thousand disks could become faulty. The extreme 3-in-10,000 imbalanced ratio poses difficulties in training a classification model. Fed with such imbalanced data, a naive classification model could attempt to judge all disks to be healthy, because in this way, it has the lowest probability of making a wrong guess. Some approaches apply data re-balancing techniques, such as over sampling and under sampling techniques, to address this challenge. These approaches help raise the recall, but at the same time could introduce a large number of false positives, which dramatically decrease the precision. In our scenario, the cost of false positives is high as the cost of VM migration is inneglectable and the cloud capacity may be affected by the false positives.

Online prediction: Existing work [9, 26] usually deals with prediction problem in a cross-validation manner. However, we found that it is inappropriate for evaluating our disk error prediction model. In cross validation, the dataset is randomly divided into training and testing set. Therefore, it is possible that the training set contains parts of future data, and testing set contains parts of past data. However, when it comes to online prediction (using historical data to train a model and predict future), training and testing data will have no time overlap. Besides, some data, especially system-level signals, are time-sensitive (their values keep changing drastically over time) or environment-sensitive (their data distribution could change due to the ever-changing cloud envi-

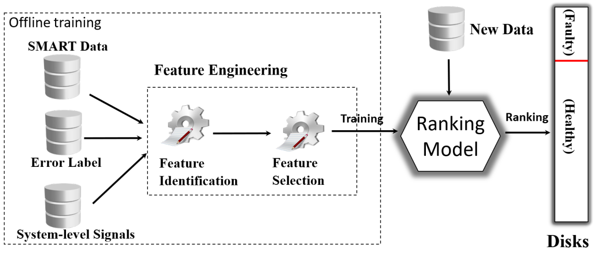


Figure 1: The overview of the proposed approach

ronment). For example, a rack encounters an outage at time t, all disks on it will experience such a change at the same time. Using cross validation, the environment-specific knowledge can spread to both training set and testing set due to random splitting. The knowledge learned from the training set could be applied to the testing set, which causes high accuracy in cross validation but poor result when evaluating new data.

Therefore, to construct an effective prediction model in practice, we should use online prediction instead of cross-validation: the future knowledge should not be known at the time of prediction.

3 Proposed Approach

In this section, we present CDEF (Cloud Disk Error Forecasting), our proposed approach that can improve service availability by predicting disk errors. Figure 2 shows the overview of CDEF. First, we collect historical data about faulty and health disks. The disk label is obtained through root cause analysis of service issues by field engineers. The feature data includes SMART data and system-level signals. We then select for training those features that are stable and predictive. Based on the selected features, we construct a cost-sensitive ranking model, which ranks the disks and identifies the top r ones that minimize the misclassification cost as the predicted faulty disks.

CDEF addresses the challenges described in the previous section by incorporating: 1) a feature engineering method for selecting stable and predictive features 2) a ranking model to increase the accuracy of cost-sensitive online prediction. We describe these two components in this section.

3.1 Feature engineering

3.1.1 Feature Identification

We collect two categories of data, SMART data and system-level signals. SMART (Self-Monitoring, Analysis and Reporting Technology) is a monitoring firmware which allows a disk drive to report data about its internal activity. Table 1 gives some of the SMART features. More information about SMART can be found in [31].

Table 1: Examples of SMART features

SMART ID	Description
S2	Start/Stop Count
S12	Power Cycle Count
S193	Load Cycle Count
S187	The number of read errors that could not be recovered using hardware ECC
S5	Count of reallocated sectors. When a read or a write operation on a sector fails, the drive will mark the sector as bad and remap (reallocate) it to a spare sector on disk.
S196	The total count of attempts to transfer data from reallocated sectors to a spare area. Unsuccessful attempts are counted as well as successful.

Table 2: The system-level signals

Signal	Description
PagingError	Windows encounters an error in creating a paging file.
FileSystem-Error	An error occurs when trying to read, write, or open a file.
DeviceReset	Device is forced to reset or shutdown.
TelemetryLoss	Telemetry data cannot be captured over a period.
DataExchange-Disabled	The data exchange integration service cannot be enabled or initialized.
VMFrozen	VM is unresponsible to communication request
Windows Event 129	A Windows event log caused by dropped requests.

In cloud systems, there are also various system-level events, which are collected periodically (typically every hour). Many of these system-level events, such as Windows events, file system operation error, unexpected telemetry loss, etc., are early signals of disk errors. Table 2 gives the descriptions of some system-level signals. For example, the *FileSystemError* is an event that is caused by disk related errors, which can be traced back to bad sectors or disk integrity corruption.

Apart from the features that are directly identified from the raw data, we also calculate some statistical features as follows:

Diff Through data analysis, we have found that the changes in a feature value over time could be useful for distinguishing disk errors. We call such a feature $Diff$.

Given a time window w, we define $Diff$ of feature x at time stamp t as follows:

$$Diff(x,t,w) = x(t) - x(t-w) \tag{1}$$

Sigma Sigma calculates the variance of attribute values within a period. Given a time window w, Sigma of attribute x at time stamp t is defined as:

$$Sigma(x,t,w) = E[(X-\mu)^2], \tag{2}$$

where $X = (x_{t-w}, x_{t-w-1}, ..., x_t)$ and $\mu = \frac{\Sigma(X)}{w}$.

Bin Bin calculates the sum of attribute values within a window w as follows:

$$Bin(x,t,w) = \sum_{j=t-w+1}^{t} x(j) \tag{3}$$

In our work, we use three different window sizes 3, 5, 7 in calculating $Diff$, Bin, and $Sigma$.

3.1.2 Feature Selection

Through the feature identification process described in the previous section, we have identified 457 features in total from SMART and system-level data. However, we have found that not all of the features can well distinguish between healthy and faulty disks, especially in the context of online prediction.

Feature selection proves very useful in selecting relevant features for constructing machine learning models. Existing feature selection methods fall into two main categories, statistical indicators (Chi-Square, Mutual Information, etc.) and machine-learning based methods like Random Forest [17]. However, in our scenario, the traditional feature selection methods cannot achieve good prediction performance because of the existence of time-sensitive and environment-sensitive features. These features carry information that are highly relevant to the training period, but may not be applicable for predicting samples in the next time period. We call this kind of features non-predictive features, meaning they have no predictive power in online prediction. Our experimental results (to be described in Section 4.3.2) show that the traditional feature selection methods lead to poor performance in our scenario.

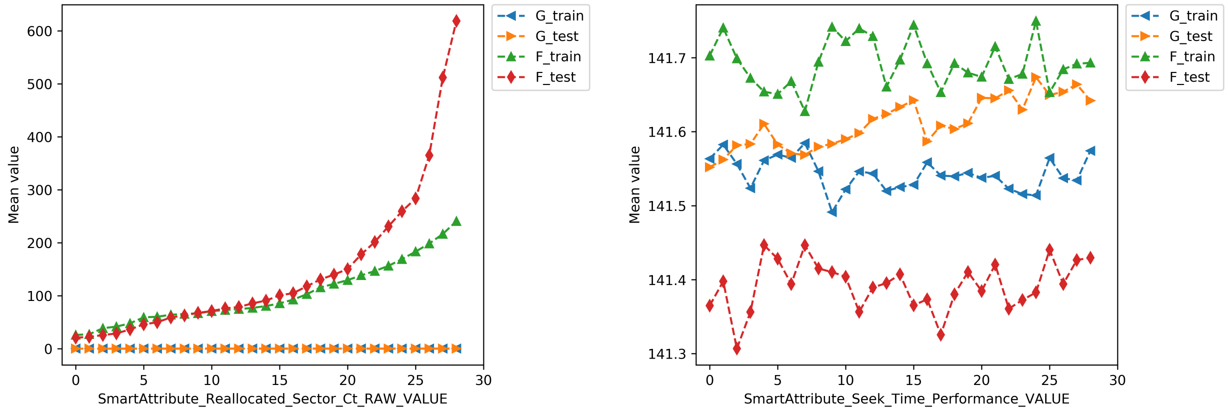
Figure 2(b) illustrates an example of a non-predictive feature *SeekTimePerformance*. Line G_train indicates the feature values of healthy disks over time in training set, and Line F_train indicates the feature values of faulty disks in the training set. Clearly, in the training set, the mean feature value of healthy disks is lower than that of faulty disks. We expect the same pattern for the same feature in the testing set (which is collected from the next time period). However, our data shows that it is not the case. In Figure 2(b), Lines G_test and F_test indicate the feature values of healthy and faulty disks over time in the testing set, respectively. Clearly, in the testing set, the mean feature value of healthy disks is higher than that of faulty disks. Therefore, the behavior of this feature is not stable. We consider this feature a non-predictive feature and not suitable for online prediction. As a comparison, Figure 2(a) shows a predictive feature *ReallocatedSectors*, from which we can see that the behavior of this feature is stable - the values of healthy disks are always close to zero and the values of faulty disks keep increasing over time, in both training and testing sets.

Algorithm 1: Prune non-predictive features

```
Input  : Training data TR with feature set $F$
         $(f_1, f_2, , , , f_m)$
Output: Reduced feature set $F'$
Split TR by time equivalently into TR1 and TR2
foreach $f_i$ in $F$ do
    // use TR1 to predict TR2, get accuracy result
    $r \leftarrow$ train(TR1) and test(TR2)
    // remove data about $f_i$ from TR, then predict
    $r_{f_i} \leftarrow$ train(TR1-$f_i$) and test(TR2-$f_i$)
    if $r_{f_i} > r$ then
        delete $f_i$ from $F$
    end
    if number of remaining features $<= \theta * m$
     then
        Break
    end
end
Return $F'$
```

To select the stable and predictive features, we perform feature selection to prune away the features that will perform poorly in prediction. The idea is to simulate online prediction on the training set. The training set is divided by time into two parts, one for training and the other for validation. If the performance on validation set gets better after deleting one feature, then the feature is deleted until the number of remaining features is less than $\theta\%$ of the total number of the features. The details are described in Algorithm 1. In our experiment, we set $\theta = 10\%$ by default, which means that the pruning process will stop if the number of remaining features is less than 10%.

At last, we re-scale the range of all selected features using zero-mean normalization as follows: $x_{zero-mean} = x - mean(X)$.



(a) Predictive features

(b) Non-predictive feature

Figure 2: An example of predictive and non-predictive feature

3.2 Cost-sensitive ranking model

Having collected features from historical data, we then construct a prediction model to predict the error-proneness of disks in the coming days. In this step, we formulate the prediction problem as a ranking problem instead of a classification problem. That is, instead of simply telling whether a disk is faulty or not, we rank the disks according to their error-proneness. The ranking approach mitigates the problem of extreme imbalanced fault data because it is insensitive to the class imbalance.

To train a ranking model, we obtain the historical fault data about the disks, and rank the disks according to their relative time to fail (i.e., the number of days between the data is collected and the first error is detected). We adopt the concept of Learning to Rank [24], which automatically learns an optimized ranking model from a large amount of data to minimize a loss function. We adopt the FastTree algorithm [28, 14], which is a form of “Multiple Additive Regression Trees” (MART) gradient boosting algorithm. It builds each regression tree (which is a decision tree with scalar values in its leave) in a step wise fashion. This algorithm is widely used in machine learning and information retrieval research.

To improve service availability, we would like to intelligently allocate VMs to the healthier disks so that these VMs are less likely to suffer from disk errors in near future. To achieve so, we identify the faulty and healthy disks based on their probability of being faulty. As most of the disks are healthy and only a small percentage of them are faulty, we select the top r results returned by the ranking model as the faulty ones. The optimal top r disks are selected in such a way that they minimize the total misclassification cost:

$$cost = Cost1 * FP_r + Cost2 * FN_r,$$

where FP_r and FN_r are the number of false positives and false negatives in the top r predicted results, respectively. Cost1 is the cost of wrongly identifying a healthy disk as faulty, which involves the cost of unnecessary live migration from the “faulty” disk to a healthy disk. Although we have very good technology for live migration, the migration process still incurs an unneglectable cost and decreases the capacity of the cloud system. Cost2 is the cost of failing to identify a faulty disk. The values of Cost1 and Cost2 are empirically determined by experts in product teams. In our current practice, due to the concerns about VM migration cost and cloud capacity, Cost1 is much higher than Cost2 (i.e., we value precision more than recall). The ratio between Cost1 and Cost2 is set to 3:1 by the domain experts based on their experience on disk error recovery. The number of false positives and false negatives are estimated through the false positive and false negative ratios obtained from historical data. The optimum r value is determined by minimizing the total misclassification cost. The top r disks are predicted faulty disks, which are high-risk disks and the VMs hosted on them should be migrated out.

4 Experiments

In this section, we evaluate the effectiveness of our approach. The aim is to answer the following research questions:

RQ1: How effective is the proposed approach in predicting disk errors?

RQ2: How effective is the proposed feature engineering method?

RQ3: How effective is the proposed ranking model?

4.1 Dataset and Setup

Dataset To evaluate the proposed approach, we collect real-world data from a large-scale Microsoft cloud service system. We use one-month data (October 2017) for training, and divide the data of November 2017 into three parts for testing. In each dataset, the ratio between healthy disks and faulty disks is around 10,000 : 3.

Setup We utilize Microsoft COSMOS [3] to store and process data collected from product teams. For ranking algorithm, we use the FastTree algorithm implemented in Microsoft AzureML [4]. We use 200 iterations in FastTree setting. The experimental evaluation is performed on a Windows Server 2012 with (Intel CPU E5-4657L v2 @2.40GHz 2.40 with 1.0 TB Memory).

4.2 Evaluation Metric

Following the existing work [23, 32, 42], we evaluate the accuracy of the proposed approach using the FPR and TPR metrics. We consider faulty disks as positive and healthy ones as negative. True Positive (TP) denotes the faulty disks that are predicted as faulty. False Positive (FP) denotes the healthy disks that are falsely predicted as faulty. True Negative (TN) denotes the healthy disks that are predicted as healthy. False Negative (FN) denotes the faulty disks that are falsely predicted as healthy. False Positive Rate (FPR) denotes the proportion of FP among all healthy disks. $FPR = FP/(FP+TN)$. True Positive Rate (TPR) denotes the proportion of TP among all faulty disks. $TPR = TP/(TP+FN)$.

We also use the ROC curve [5] that plots TPR (True Positive Rate) versus FPR (False Positive Rate), and compute the Area Under Curve (AUC). Following the related work [23, 29], we compute the TPR value when FPR is 0.1%, which indicates how good an algorithm can predict faulty disks under a high precision requirement.

4.3 Results

4.3.1 RQ1: How effective is the proposed approach in predicting disk errors?

We evaluate the effectiveness of the proposed CDEF approach on all three datasets. We also compare CDEF with the Random Forest and SVM based methods proposed in the related work on disk failure prediction [26, 32]. These methods use the Random Forest or SVM classifiers to classify disks based on the SMART data. We treat them as baseline methods in this experiment.

The experimental results are shown in Figure 3. The diagonal lines indicate the accuracy obtained by Random Guess (meaning random prediction with 50% probability). The results show that CDEF outperforms the baseline methods consistently under different FPR/TPR ratios on all datasets. For example, on Dataset 1, the AUC values for our approach is 0.93, while the AUC value for Random Guess, Random Forest, and SVM is 0.5, 0.85, and 0.53, respectively.

We evaluate the effectiveness of the proposed ranking approach in terms of misclassification cost and the TPR value (when FPR is 0.1%). The misclassification cost is obtained as: cost= Cost1*FP+Cost2*FN, where Cost1 and Cost2 are set to 3 and 1 respectively by the product team. Table 3 shows the results. Clearly, CDEF obtains better results than the other two methods. The TPR value is 36.50%, 41.09%, and 29.67% on Dataset 1, 2, and 3, respectively. CDEF is also cost-effective. In average, CDEF achieves around 187.92% cost reduction than Random Forest, and 10.13% cost reduction than SVM. SVM has low cost because SVM is accurate in predicting healthy disks and induces less false positives. But SVM performs worse in predicting faulty disks and induces low TPR.

In summary, the experimental results show that CDEF is effective in predicting disk errors. This is because of two reasons: the proposed feature engineering method and the proposed ranking model. We will show the effectiveness of these two methods in the following RQs.

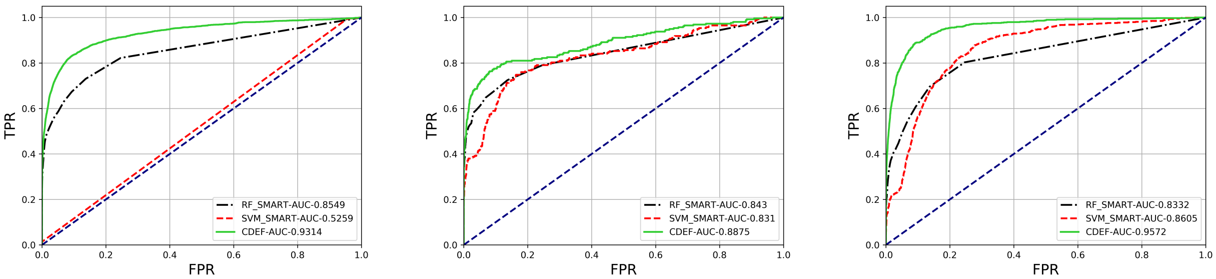
Table 3: Experimental results of CDEF on three datasets

	CDEF		RandomForest		SVM	
	Cost	TPR	Cost	TPR	Cost	TPR
Dataset 1	2508	36.50%	3157	30.51%	2907	15.51%
Dataset 2	234	41.09%	1211	34.11%	258	21.71%
Dataset 3	760	29.67%	1675	18.81%	792	7.20%

4.3.2 RQ2: How effective is the proposed feature engineering method?

In our work, we propose to use system-level signals in disk error prediction. We also propose a feature selection method to select the predictive features for model training. In this RQ, we evaluate the effectiveness of our proposed feature engineering method. We experiment with three feature engineering methods: S (traditional SMART-based features), S+A (SMART and system-level signals), and S+A+F (SMART and system-level signals with feature selection, which is used in CDEF). All other experimental settings remain the same.

The results are shown in Figure 4. We can see that the results achieved by incorporating system-level signals outperform those achieved by SMART alone on all the three datasets. Furthermore, by incorporating SMART and system-level signals with feature selection, we can obtain the best results on all the three datasets. In average, the TPR value (when FPR is 0.1%) is 27.6%, 30.3%, and 35.8%, for S, S+A, and S+A+F, respectively. These results confirm the effectiveness of the proposed feature engineering methods.



(a) Dataset 1 (b) Dataset 2 (c) Dataset 3

Figure 3: ROC of comparative methods

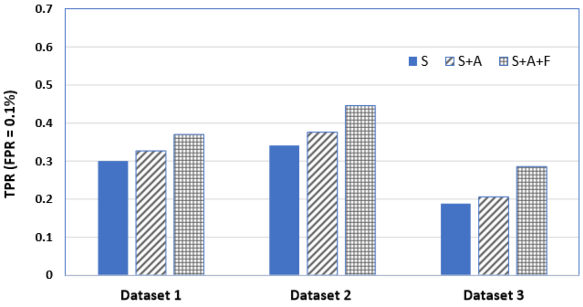


Figure 4: Evaluation results with different features
S: traditional SMART-based features; S+A: SMART and system-level signals; S+A+F: SMART and system-level signals with feature selection.

We also evaluate the effectiveness of CDEF using the features selected by the proposed feature selection method and the features selected by conventional feature selection methods Chi-Square, Mutual Information, and Random Forest [17, 21]. The results are given in Figure 5, which shows that the proposed feature selection method outperforms the conventional feature selection methods on all datasets.

4.3.3 RQ3: How effective is the proposed ranking model?

In our work, we propose to use a cost-sensitive ranking method to rank the disks and then select the top *r* disks as faulty ones by minimizing the total misclassification cost. In this RQ, we evaluate the effectiveness of the proposed ranking approach.

To perform classification for imbalanced data, one common approach is to apply the over-sampling technique SMOTE [10] to balance the training data for model construction. The other approach is weighted classification, which is essentially cost-sensitive learning [12] that learns from extremely imbalanced data and assigns a larger weight to minority class. The weight is usually set inversely to the sample portion. In our experiment, we compare the proposed cost-sensitive ranking method with these two approaches. To better evaluate the accuracy of the proposed method, we also compare with the random guess method.

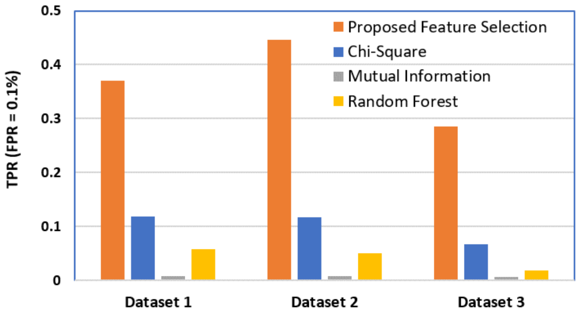


Figure 5: The comparison between the proposed feature selection method and existing methods

We evaluate the effectiveness of the proposed ranking approach in terms of misclassification cost. The proposed cost-sensitive ranking model achieves the minimum cost among all comparative methods on all datasets. For example, on Dataset 2, the misclassification cost obtained by our model is 234, while cost obtained by Random Guess, weighted classification, and classification with SMOTE are much higher (1146662, 717, and 7812, respectively).

We also evaluate the effectiveness of the proposed ranking approach in terms of TPR and FPR values. Figure 6 shows the ROC curves achieved by the comparative methods. Table 4 shows the TPR values when FPR is 0.1%, achieved by different methods on all the datasets. Clearly, our cost-sensitive ranking method achieves the best accuracy values. For example, on Dataset 2, the TPR value (when FPR is 0.1%) achieved by our model is 41.09%, while the values achieved by Random Guess, weighted classification, and classification with SMOTE are much lower (0.1%, 27.91%, and 27.94%, respectively). The AUC value achieved by our model is 88.75%, while the values achieved by Random Guess, weighted classification, and classification with SMOTE are 0.5%, 84.22%, and 83.56%, respectively.

In summary, the experimental results confirm the effectiveness of the proposed cost-sensitive ranking model.

4.4 Discussions of the Results

In our work, we do not use cross-validation to build and evaluate the proposed approach. Instead, we do online prediction - using the data before a certain date to train the model and use the data after the date to test the model. Existing work on failure prediction such as [9, 26] uses cross-validation to evaluate their machine-learning based models. In our scenario, cross-validation can lead to much better results than online prediction, as shown in Figure 7. For example, on Dataset 1, using cross-validation we can obtain TPR value of 91.64% (when FPR is 0.1%), while using online prediction the TPR value is only 36.50%. However, our experiences show that cross-validation may not always reflect the actual effectiveness of a prediction model. Online prediction should be used in practice.

In cross validation, the dataset is randomly divided into training and testing sets. Therefore, it is possible that the training set contains parts of future data, and the testing set contains parts of past data. However, in real-world online prediction, training and testing sets are divided by time. The past data is used to train the model and the future data is used to test the model.

The gap is magnified when there are time-sensitive features and environment-sensitive features. In disk error prediction, some features have temporal nature and their values vary drastically over time. Some features may be easily affected by environmental changes to the cloud system. For example, the disks on the same rack or the same motherboard encounter similar attribute changes caused by unstable voltage. However, such changes may not happen before the time of prediction. Using cross-validation we may utilize the knowledge that should not be known at the time of prediction, thus obtaining better evaluation results. Therefore, cross-validation is not suitable for evaluating our model in practice. The problem of cross-validation in evaluating an online prediction model is also observed by others [36].

4.5 Threats to Validity

We have identified the following threats to validities:

Subject systems: In our experiments, we only collect data from one cloud service system of one company. Therefore, our results might not be generalizable to other systems. However, the system we studied is a typical, large-scale cloud service system, from which sufficient data can be collected. Furthermore, we have applied our approach in the maintenance of the cloud system. In future, we will reduce this threat by evaluating CDEF on more subject systems and report the evaluation results.

Data noise: After a disk is identified to be faulty, it could be sent to repair. After that, some disks could be returned and used again. Therefore, a small degree of noise may exist in the labeling of a disk.

Evaluation metrics: We used the FPR/TPR metrics to evaluate the prediction performance. These metrics have been widely used to evaluate the effectiveness of a disk fault prediction mode [32]. Prior work [38] points out that a broader selection of metrics should be used in order to maximize external validity. In our future work, we will reduce this threat by experimenting with more evaluation measures such as Recall/Precision.

5 Lessons Learned from Practice

We have successfully applied CDEF to the maintenance of Microsoft Azure, which is a large-scale cloud service system that allows IT professionals to build, deploy, and manage applications. The cloud service achieves global scale on a worldwide network of data centers across many regions. Due to the unreliable nature of the underlying commodity hardware, various issues occur in Azure every day. Without proper handling of these issues, Azure service availability could be seriously affected. We found disk error is the most severe one among all hardware issues.

CDEF is currently used by Azure to preferentially select healthier disks for VM allocation and live migration. After deploying CDEF, in average, we successfully saved around 63k minutes of VM downtime per month. Note that 99.999% service availability means that only 26 seconds per month of VM downtime is allowed. Therefore, CDEF has significantly improved service availability of Microsoft Azure.

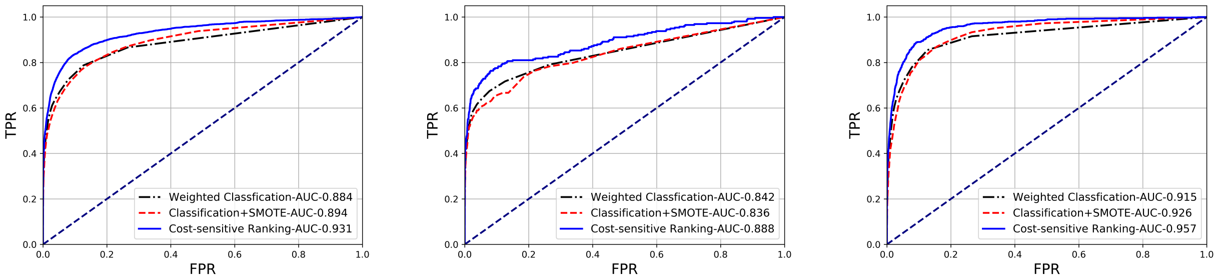
Currently the training is performed daily over the past 90-day data, and keeps a moving window of 90 days. The cutting point *r* in the ranking model is set along with the training process. When a disk is predicted as faulty, we mark the host node unallocable and trigger live migration process. We also run disk stress test on the predicted disks before they are taken out for replacement.

We have learned the following lessons from our industrial practice:

- **Continuous training**. Many factors could affect the distribution of disk error data, such as bugs in OS driver/firmware, workload on clusters, platform maintenance, etc. A model trained in the past will not always work in the future. Therefore, we build a continuous training pipeline. For every predicted disk error, we also let the disk go through a disk

Table 4: The cost and TPR values (when FPR is 0.1%) achieved by the proposed cost-sensitive ranking model

	Random Guess		Cost-sensitive ranking		Weighted Classfication		Classification+SMOTE	
	Cost	TPR	Cost	TPR	Cost	TPR	Cost	TPR
Dataset 1	1447986	0.1%	2508	36.50%	2910	26.52%	9442	24.63%
Dataset 2	1146662	0.1%	234	41.09%	717	27.91%	7812	27.94%
Dataset 3	1446929	0.1%	760	29.67%	1234	17.42%	8239	17.68%



(a) Dataset 1

(b) Dataset 2

(c) Dataset 3

Figure 6: ROC of cost-sensitive ranking and classification

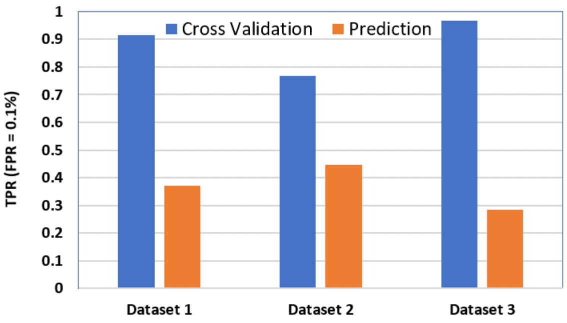


Figure 7: Evaluation results - cross validation vs. online prediction

stress test to check if it is really faulty. This forms a continuous feedback loop between disk error prediction and disk stress test.

- **Cost-effectiveness**. Prediction alone may not make much impact if the cost of recovery operation is really high (because the cost of leaving the host node as it is might be cheaper than the cost of taking the recovery operation). Furthermore, the cost to recover a node with one VM on top is much cheaper than the cost of recovery with 10 VMs in terms of VM availability. Thus, the cost of recovery could vary depending on the state of the host node, the recovery operation, etc. The prediction could be even more useful if we can better estimate the cost.

- **Faulty disks will get even worse.** Our experience shows that before a disk completely fails, it may already start emitting errors that affect upper-layer applications and services. That is why incorporating the system-level signals is better than using SMART alone. We found that disk errors, in average, occur 15.8 days earlier than complete disk failure. Our experience also shows that, before completely fails, the status of a disk will actually get worse over time. For example, for faulty disks, the value of the feature *ReallocatedSectors* increases by 3 times during the last week of its operation. The value of system-level signal *DeviceReset* even increases by 10 times during the same period. This finding confirms our intention to detect disk error earlier before it makes severe impact on application usage.

6 Related Work

6.1 Disk Failure Prediction

There are a large amount of related work on predicting disk failures. For example, BackBlaze publishes quarterly report [6] for users to keep track of reliability of popular hard drives in the market. Most of the modern hard drives support Self-Monitoring, Analysis and Reporting Technology (SMART), which can monitor internal attributes of individual drives. SMART is used by some manufacturers to predict impending drive failure by simple threshold-based method [31, 34].

As the prediction performance of the thresholding algorithm is disappointing, researchers have proposed various machine learning models for predicting disk failures. For example, Zhu et al. [42] predicted disk failure based on raw SMART attributes and their change rates, and neural network and SVM model are applied. Ganguly et al. [16] utilized SMART and hardware-level features such as node performance counter to predict disk failure. Ma et al. [25] investigate the impact of disk failures on RAID storage systems and designed RAIDShield to predict RAID-level disk failures.

Tan et al. [37] proposed an online anomaly prediction method to foresee impending system anomalies. They applied discrete-time Markov chains (DTMC) to model the evolving patterns of system features, then used tree-augmented naive Bayesian to train anomaly classifier. Dean et al. [11] proposed an Unsupervised Behavior Learning (UBL) system, which leverages an unsupervised method Self Organizing Map to predict performance anomalies. Wang et al. [41] also proposed an unsupervised method to predict drive anomaly based on Mahalanobis distance. There are also other work [19, 40] in online machine learning [7], which aims to update the best predictor at each step for steaming data (as opposed to batch learning techniques). While our "online prediction" focuses on the prediction workflow: always using a batch of historical data to predict the future (as opposed to cross-validation). Furthermore, unlike [37], we deal with the evolving features by proactively selecting the consistently predictive features. Unlike [11, 41] that can be used even when label data is difficult to get, we adopt a supervised method as we have quality labeled data. We will compare our method with unsupervised-learning based methods in our future work.

For feature selection, Botezatu et al. [9] selected the most relevant features based on statistical measures. Gaber et al. [15] used machine learning algorithms to extract features representing the behavior of the drives and predict the failure of the drives. However, these feature selection methods are not able to prune non-predictive features in online prediction scenario.

All these related work focus on disk failure prediction based on SMART and other hardware-level attributes. While our work focuses on predicting disk errors that affect the availability of virtual machines, before complete disk failure happens. We incorporate both SMART and system-level signals, which proves better than using SMART data alone. Also, most of the related work evaluate their prediction model in a cross validation manner, which is not appropriate in real-world practice. In our work, we perform online prediction and propose a novel algorithm to select stable and predictive features.

6.2 Failures in Cloud Service Systems

Although tremendous effort has been made to maintain high service availability, in reality, there are still many unexpected system problems caused by software or platform failures (such as software crashes, network outage, misconfigurations, memory leak, hardware breakdown, etc.). There have been some previous studies in the literature on failures of a data center. For example, Ford et al. studied [13] the data availability of Google distributed storage systems, and characterized the sources of faults contributing to unavailability. Their results indicate that cluster-wide failure events should be paid more attention during the design of system components, such as replication and recovery policies. Vishwanath and Nagappan [39] classified server failures in a data center and found that 8% of all servers had at least one hardware incident in a given year. Their studies could be helpful to reduce the hardware faults, especially the networking faults. Huang et al. [22] also found that the major availability breakdowns and performance anomalies we see in cloud environments tend to be caused by subtle underlying faults, i.e., gray failure rather than fail-stop failure. The above-mentioned related work shows that failures in cloud systems can be triggered by many software or hardware issues. In our work, we only focus on disk error prediction. In particular, disk errors can be also seen as a form of *gray failures*: the system's failure detectors may not notice them even when applications are afflicted by them.

7 Conclusion

Disk error is one of the most important reasons that cause service unavailability. In this paper, we propose CDEF, an online disk error prediction approach that can predict disk errors proactively before they cause more severe damage to the cloud system. We collect both SMART and system-level signals, perform feature engineering, and develop a cost-sensitive ranking model. We evaluate our approach using real-world data collected from a cloud system. The results confirm that the proposed approach is effective and outperforms related methods. The ability to predict faulty disks enables the live migration of existing virtual machines and allocation of new virtual machines to the healthy disks, thus improving service availability. We have also successfully applied the proposed approach to Microsoft Azure.

There are many viable ways of extending this work. We have applied our approach to hard disk drives in production. In the future, we will apply it to other disk types such as Solid State Drive. We will also explore the synergy between disk error prediction and other cloud failure detection techniques such as [22], and propose an integrated solution to service availability improvement.

8 Acknowledgements

Our special thanks go to Girish Bablani, Gil Shafriri, Aaron Colling, Zheng Mu, Brijesh Ramachandran, John Kim, Sandeep Bhadsavle, Ashish Munjal, and Nisarg Sheth for making this research impactful for Azure service. We especially thank our shepherd Xiaohui Gu for the constructive comments.

References

[1] http://www.distributed-generation.com.

[2] https://www.backblaze.com/b2/hard-drive-test-data.html.

[3] https://azure.microsoft.com/en-au/services/cosmos-db/.

[4] https://studio.azureml.net/.

[5] https://en.wikipedia.org/wiki/Receiver_operating_characteristic.

[6] https://www.backblaze.com/blog/hard-drive-failure-rates-q3-2017/.

[7] http://www.distributed-generation.com.

[8] Bairavasundaram, L. N., Goodson, G. R., Pasupathy, S., and Schindler, J. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2007), SIGMETRICS '07, ACM, pp. 289–300.

[9] Botezatu, M. M., Giurgiu, I., Bogojeska, J., and Wiesmann, D. Predicting disk replacement towards reliable data centers. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016), ACM, pp. 39–48.

[10] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res. 16*, 1 (June 2002), 321–357.

[11] Dean, D. J., Nguyen, H., and Gu, X. UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th international conference on Autonomic computing* (2012), ACM, pp. 191–200.

[12] Domingos, P. Metacost: A general method for making classifiers cost-sensitive. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining* (1999), ACM, pp. 155–164.

[13] Ford, D., Labelle, F., Popovici, F., Stokely, M., Truong, V.-A., Barroso, L., Grimes, C., and Quinlan, S. Availability in globally distributed storage systems. In *Proc. OSDI* (2010).

[14] Friedman, J. H. Greedy function approximation: A gradient boosting machine. *Annals of Statistics 29* (2000), 1189–1232.

[15] Gaber, S., Ben-Harush, O., and Savir, A. Predicting hdd failures from compound smart attributes. In *Proceedings of the 10th ACM International Systems and Storage Conference* (2017), SYSTOR '17, ACM, pp. 31:1–31:1.

[16] Ganguly, S., Consul, A., Khan, A., Bussone, B., Richards, J., and Miguel, A. A practical approach to hard disk failure prediction in cloud platforms: Big data model for failure management in datacenters. In *Big Data Computing Service and Applications (BigDataService), 2016 IEEE Second International Conference on* (2016), IEEE, pp. 105–116.

[17] Genuer, R., Poggi, J.-M., and Tuleau-Malot, C. Variable selection using random forests. *Pattern Recognition Letters 31*, 14 (2010), 2225 – 2236.

[18] Goldszmidt, M. Finding soon-to-fail disks in a haystack. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems* (Berkeley, CA, USA, 2012), HotStorage'12, USENIX Association, pp. 8–8.

[19] Gu, X., and Wang, H. Online anomaly prediction for robust cluster systems. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on* (2009), IEEE, pp. 1000–1011.

[20] Gunawi, H. S., Suminto, R. O., Sears, R., Golliher, C., Sundararaman, S., Lin, X., Emami, T., Sheng, W., Bidokhti, N., McCaffrey, C., et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems.

[21] Guyon, I., and Elisseeff, A. An introduction to variable and feature selection. *J. Mach. Learn. Res. 3* (Mar. 2003), 1157–1182.

[22] Huang, P., Guo, C., Zhou, L., Lorch, J. R., Dang, Y., Chintalapati, M., and Yao, R. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2017), HotOS '17, ACM, pp. 150–155.

[23] Li, J., Ji, X., Jia, Y., Zhu, B., Wang, G., Li, Z., and Liu, X. Hard drive failure prediction using classification and regression trees. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (June 2014), pp. 383–394.

[24] Liu, T.-Y. Learning to rank for information retrieval. *Found. Trends Inf. Retr. 3*, 3 (Mar. 2009), 225–331.

[25] Ma, A., Traylor, R., Douglis, F., Chamness, M., Lu, G., Sawyer, D., Chandra, S., and Hsu, W. Raidshield: Characterizing, monitoring, and proactively protecting against disk failures. *ACM Trans. Storage 11*, 4 (Nov. 2015), 17:1–17:28.

[26] Mahdisoltani, F., Stefanovici, I., and Schroeder, B. Proactive error prediction to improve storage system reliability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 391–402.

[27] Meza, J., Wu, Q., Kumar, S., and Mutlu, O. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2015), SIGMETRICS '15, ACM, pp. 177–190.

[28] Microsoft. Machine learning fast tree, https://docs.microsoft.com/en-us/machine-learning-server/python-reference/microsoftml/rx-fast-trees, 2017.

[29] Murray, J. F., Hughes, G. F., and Kreutz-Delgado, K. Machine learning methods for predicting failures in hard drives: A multiple-instance application. *Journal of Machine Learning Research 6*, May (2005), 783–816.

[30] Patterson, D. A., Gibson, G., and Katz, R. H. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1988), SIGMOD '88, ACM, pp. 109–116.

[31] Pinheiro, E., Weber, W.-D., and Barroso, L. A. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2007), FAST '07, USENIX Association, pp. 2–2.

[32] Pitakrat, T., van Hoorn, A., and Grunske, L. A comparison of machine learning algorithms for proactive hard disk drive failure detection. In *Proceedings of the 4th International ACM Sigsoft Symposium on Architecting Critical Systems* (New York, NY, USA, 2013), ISARCS '13, ACM, pp. 1–10.

[33] PonemonInstitute. Cost of data center outages, 2016. https://planetaklimata.com.ua/instr/Liebert_Hiross/Cost_of_Data _Center_Outages_2016_Eng.pdf.

[34] Schroeder, B., and Gibson, G. A. Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you? In *FAST* (2007), pp. 1–16.

[35] Schroeder, B., Lagisetty, R., and Merchant, A. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 67–80.

[36] Tan, M., Tan, L., Dara, S., and Mayeux, C. Online defect prediction for imbalanced data. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2* (Piscataway, NJ, USA, 2015), ICSE '15, IEEE Press, pp. 99–108.

[37] Tan, Y., and Gu, X. On predictability of system anomalies in real world. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on* (2010), IEEE, pp. 133–140.

[38] Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. Comments on 'researcher bias: The use of machine learning in software defect prediction'. *IEEE Transactions on Software Engineering 42*, 11 (Nov 2016), 1092–1094.

[39] Vishwanath, K. V., and Nagappan, N. Characterizing cloud computing hardware reliability. In *SOCC* (New York, NY, USA, 2010), ACM, pp. 193–204.

[40] Wang, C., Talwar, V., Schwan, K., and Ranganathan, P. Online detection of utility cloud anomalies using metric distributions. In *Network Operations and Management Symposium (NOMS), 2010 IEEE* (2010), IEEE, pp. 96–103.

[41] Wang, Y., Miao, Q., Ma, E. W. M., Tsui, K. L., and Pecht, M. G. Online anomaly detection for hard disk drives based on mahalanobis distance. *IEEE Transactions on Reliability 62*, 1 (March 2013), 136–145.

[42] Zhu, B., Wang, G., Liu, X., Hu, D., Lin, S., and Ma, J. Proactive drive failure prediction for large scale storage systems. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)* (May 2013), pp. 1–5.

RAFI: Risk-Aware Failure Identification to Improve the RAS in Erasure-coded Data Centers

Juntao Fang[‡], Shenggang Wan[†], and Xubin He[§]
[†]*School of Computer Science and Technology, Huazhong University of Sci. and Tech., China*
[‡]*Wuhan National Laboratory for Optoelectronics, Huazhong University of Sci. and Tech., China*
[§]*Department of Computer and Information Sciences, Temple University, USA*
{yydfjt, sgwan}@hust.edu.cn, xubin.he@temple.edu

Abstract

Data reliability and availability, and serviceability (RAS) of erasure-coded data centers are highly affected by data repair induced by node failures. Compared to the recovery phase of the data repair, which is widely studied and well optimized, the failure identification phase of the data repair is less investigated. Moreover, in a traditional failure identification scheme, all chunks share the same identification time threshold, thus losing opportunities to further improve the RAS.

To solve this problem, we propose *RAFI*, a novel risk-aware failure identification scheme. In RAFI, chunk failures in stripes experiencing different numbers of failed chunks are identified using different time thresholds. For those chunks in a high risk stripe (a stripe with many failed chunks), a shorter identification time is adopted, thus improving the overall data reliability and availability. For those chunks in a low risk stripe (one with only a few failed chunks), a longer identification time is adopted, thus reducing the repair network traffic. Therefore, the RAS can be improved simultaneously.

We use both simulations and prototyping implementation to evaluate RAFI. Results collected from extensive simulations demonstrate the effectiveness and efficiency of RAFI on improving the RAS. We implement a prototype on HDFS to verify the correctness and evaluate the computational cost of RAFI.

1 Introduction

In large-scale erasure-coded data centers, node failures are the norm rather than the exception [1]. Those frequent node failures can result in numerous chunk failures (a chunk is the basic unit to organize data). The *RAS* (*R*eliability, *A*vailability, and *S*erviceability) of data centers are highly affected by repairing those failed chunks, which is known as *data repair*. Many solutions [2–19] are proposed to improve the RAS, i.e., reduce data loss, unavailability, and repair network traffic (a typical repair cost), through optimizing the data repair. However, existing solutions typically focus on the *recovery phase*, which is from the time when a chunk failure is identified to the time when the failed chunk is recovered. In contrast, the *identification phase*, which is from the time when a chunk failure occurs to the time when the chunk failure is identified, has not been explored yet. Consequently, the potential to further improve the RAS is not fully explored.

Traditionally, the failure identification of a chunk depends on the failure identification of its host node. When a node fails, its failure is not identified until a certain time threshold. When the node failure is identified, the failures of all the chunks on that node are identified, and the states of those chunks transition to *lost*. In summary, all chunks share the same time threshold with nodes in a traditional failure identification (TFI) scheme.

Under the TFI scheme, it is hard to simultaneously improve the RAS through adjusting the time threshold. On one hand, higher data reliability and availability could be achieved by lowering the failure identification time threshold, because of the shortened data repair time. On the other hand, the data center might suffer from increasing repair network traffic, because more transient node failures might be identified. In contrast, by increasing the failure identification time threshold, the repair network traffic could be reduced but the data reliability and availability might be suffer.

In this paper, we posit that the RAS can be simultaneously improved through optimizing the identification phase. This is rooted in the following dedicated observation on stripes. Each stripe consists of data chunks and parity chunks generated from those data chunks. A stripe is the basic unit for ensuring data reliability and availability. According to the number of failed chunks in a stripe, failed stripes can be classified into two types. One is a stripe which has many failed chunks, e.g., by default two or more failed chunks in a stripe with three parity chunks. This type of failed stripes is referred to as a high risk stripe. The other is referred to as a low risk stripe, which has a few failed chunks, e.g., by default one failed chunk in a stripe with three parity chunks. The more failed chunks a stripe has, the lower the data reliability and availability of the stripe are. Hence, most of the data loss and unavailability occur in high risk stripes. On the other hand, low risk stripes are much more common than high risk stripes, and thus induce most of the repair network traffic.

There already exist solutions that improve the RAS in the failure recovery phase, are rooted in being aware of the risk of stripes, e.g., prioritizing the recovery of the chunks in the stripes with multiple lost chunks [3, 7], or canceling the recovery of the chunks in the stripes with a few lost chunks [14]. Inspired by these approaches, we propose a novel *R*isk-*A*ware *F*ailure *I*dentification scheme, named *RAFI*, to improve the RAS of erasure-coded data centers. More specifically, RAFI is aware of not only lost chunks, which are focused on the traditional risk-aware wisdom, but also *unidentified failed* chunks, whose failure has not been identified yet. The key principle of RAFI is that the more failed chunks a stripe has, the shorter failure identification time threshold those chunks take. As a result, the aforementioned conflict between the data reliability and availability, and the repair network traffic is resolved, and the RAS are improved simultaneously.

We make the following contributions in this paper.

(1) We propose a risk-aware failure identification scheme RAFI to simultaneously improve the RAS of erasure-coded data centers. By deploying RAFI, a chunk failure is identified through multiple independent identification thresholds. Therefore, for chunks in high risk stripes, their failure identification is expedited, thus improving the data reliability and availability. For chunks in low risk stripes, their failure identification is postponed, thus reducing the repair network traffic. As a result, the RAS are improved simultaneously.

(2) A simulator is developed to verify our RAFI. The simulation results demonstrate that RAFI is very effective and efficient. For example, cooperating with all types of the state-of-the-art optimizations on the failure recovery phase, RAFI can further improve the data reliability by a factor of 9.3, and reduce the data unavailability and repair network traffic by 43% and 36%, respectively, at the cost of degraded reads increased by 1.7%.

(3) A prototype of RAFI is implemented in HDFS to verify the correctness and computational cost of our RAFI. The experimental results demonstrate that, in the worst case scenario, the computational cost of RAFI is still negligible.

The rest of this paper is organized as follows: Section 2 presents a model to analyze the relevance among the data reliability, repair network traffic, and failure identification. In Section 3, we give the design of RAFI. The results of prototype experiments and simulations are illustrated in Section 4 and 5, respectively. Section 6 reviews related work on optimizing the failure recovery phase, and Section 7 concludes the paper.

2 Background and Motivation

In this section, we first define the terms used in this paper. Then, we review the background materials of erasure-coded data centers, and summarize the existing methods to improve the RAS. Finally, we illustrate our motivation to propose RAFI.

2.1 Terms

Some terms to facilitate our discussion are summarized as follows.

A failed node: a node whose heartbeats have been lost. When a node fails, its heartbeat is lost immediately and it becomes a failed node. In TFI, the failure of a node is not identified until its heartbeats have been lost for over a certain time threshold.

A failed chunk: a chunk whose host node fails. When a node fails, all chunks on that node become failed. A failed chunk can be further classified into an unidentified failed chunk and a lost chunk as described below.

An unidentified failed chunk: a failed chunk whose failure has not been identified yet. Between the chunk failure occurs and that failure is identified, the chunk is treated as unidentified failed.

A lost chunk: a failed chunk whose failure is identified. After the failure of a chunk is identified, the chunk is treated as lost.

S^i and S^{i+}: a stripe S^i is a stripe with i lost chunks, and a stripe S^{i+} is a stripe with i and more lost chunks.

2.2 Erasure-coded Data Centers

To tolerate node failures, data redundancy techniques are usually deployed in data centers. Traditional data redundancy techniques, e.g., replication, suffer from high spatial cost. Hence, erasure coding techniques (e.g., Reed-Solomon coding) which have a much lower spatial cost compared to replication techniques, are widely used in data centers [7, 12, 20, 21].

To apply the erasure coding in data centers, data is first divided into fixed size data chunks. Then, parity chunks are generated from those data chunks. To prevent data loss or unavailability from node failures, all those data and parity chunks together form a stripe and are distributed to different nodes.

Table 1: Methods to Improve the RAS

	Time Threshold ↓	Recovery Penalty Factor ↑	Network Bandwidth ↑	Queue Time ↓
Reliability/Availability	↑	↑	↑	↑
Repair Network Traffic	↑	↓	→	→

Node failures are monitored through frequent heartbeats, e.g., every 3 seconds [3]. However, a node failure is not immediately identified when the heartbeats are lost, because most node failures are transient and those failed nodes can come back in a short period, e.g., 10 minutes [20]. In order to reduce the repair network traffic, only when the heartbeats have been lost over a certain time threshold, e.g., 15 minutes [20] or 30 minutes [7], a node failure is identified (a misidentification occurs if the node comes back).

Traditionally, when a node failure is identified, all the chunk failures due to that node failure are treated as identified failures. Surviving data and parity chunks (on other nodes) of the lost chunks would be fetched to repair those lost chunks (data repair), thus ensuring the data availability and reliability.

2.3 Methods to Improve the RAS

It is cost-effective to improve the RAS by optimizing the data repair process. Many solutions are proposed following this way which are explained below and also summarized in Table 1.

(1) Decreasing the time threshold reduces the repair time, and thus improves the reliability; however, it increases the repair network traffic;

(2) In erasure-coded data centers, multiple available chunks are transmitted over the network to recover lost chunks in the stripe. Recovery penalty factor is a factor which is between the amount data transmitted for recovering a stripe S^i and the size of a chunk. Decreasing the recovery penalty factor [2, 4, 5, 7–13, 16, 17, 22, 23] reduces the repair time, and thus improves the reliability; in the meanwhile, it reduces the repair network traffic;

(3) Increasing the network bandwidth [6, 24–26] of each storage node reduces the repair time, and thus improves the reliability; in the meanwhile, the repair network traffic stays almost the same.

(4) The queue time (waiting for recovery) of failed stripes is affected by recovery schemes. Giving high priority to S^i $(i > 1)$ [7, 27], the queue time of S^i $(i > 1)$ is decreased, and thus the reliability is improved; in the meanwhile, this method has little effect on the repair network traffic.

According to the above analysis and simulation results demonstrated in Figure 9a in Section 5.3, the RAS cannot be improved simultaneously by adjusting the failure identification time threshold. Therefore, a novel risk-aware failure identification scheme RAFI is proposed to explore the huge potential of simultaneously improving the RAS within the failure identification phase.

2.4 Motivation

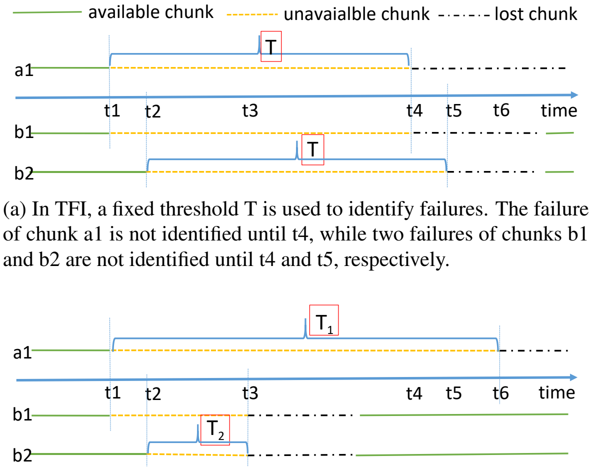
When some nodes fail, many stripes are affected, i.e., have failed chunks. Due to the randomized chunk layout, only a small fraction of those affected stripes have many failed chunks, and the remaining affected stripes only have a few failed chunks. Hence, most repair network traffic is induced by repairing the latter type of stripes.

On the other hand, the failure identification time of an arbitrary affected stripe having i failed chunks is equal to the failure identification time of its ith failed chunk, i.e., all the affected stripes share the same failure identification time. The stripes with many lost chunks usually entitle high recovery priority, i.e., a short queuing time. Hence, the repair time of those stripes are usually dominated by the failure identification time. In contrast, the stripes with a few identified failed chunks usually have a long queuing time. Hence, the repair time of those stripes are usually dominated by the recovery time.

If the failure identification of those two types of stripes can be handled separately, the RAS of data centers can be improved simultaneously. More specifically, for the stripes having many failed chunks, we tune down the failure identification time threshold of those failed chunks, and thus improving the data availability and reliability at the cost of slightly increasing repair network traffic. For the stripes having a few failed chunks, we tune up the failure identification time threshold of those failed chunks, and thus reducing the repair network traffic without significantly reducing data reliability and availability. More importantly, the benefit induced by the above two operations would be dominant compared to the associated cost. Hence, the RAS of data centers can be improved simultaneously.

3 RAFI: Design and Analysis

In this section, we first present the design of RAFI; followed by a discussion on the benefit and cost of deploying RAFI.



(a) In TFI, a fixed threshold T is used to identify failures. The failure of chunk a1 is not identified until t4, while two failures of chunks b1 and b2 are not identified until t4 and t5, respectively.

(b) In RAFI, the failure of chunk a1 is identified through the threshold T_1 at t6, which is later than t4. On the other hand, the failures of chunks b1 and b2 are identified through the threshold T_2 at t3, which is ahead of t4 and t5.

Figure 1: Identification of chunk failures using TFI and RAFI. We use three sample chunks, where a1 is a random chunk of a stripe A while b1 and b2 are two random chunks of a stripe B. Assume chunk a1 fails at time t1 while chunks b1 and b2 fail at t1 and t2, respectively.

3.1 Design of RAFI

As we discussed above, the key problem of the traditional failure identification (TFI) scheme is that all chunks share the same failure identification time threshold. To simultaneously improve the RAS, we propose *RAFI* to identify chunk failures according to the risk level of their host stripes and apply different time thresholds accordingly. More specifically, dedicated chunk failure identification time thresholds are set for stripes in different risk levels, which are determined by the *total failed chunks* in the stripes. For chunks in low risk stripes, long failure identification time thresholds are adopted, thus reducing the repair cost. For chunks in high risk stripes, short failure identification time thresholds are adopted, thus improving the data reliability and availability. As a result, the RAS are simultaneously improved.

In summary, the key design principle of RAFI is that the more failed chunks a stripe has, the shorter failure identification threshold those chunks take. For a failed chunk in a stripe with i failed chunks, there are at most i identification thresholds to identify the failure of this chunk, and the jth ($0 < j \leq i$) identification threshold is described as follows. If there are j failed chunks and the failure durations of these j failed chunks are all longer than a preset time threshold T_j, all these j chunk failures are identified and these chunks are denoted as lost immediately. The state of an unidentified failed chunk in these j chunks transitions to lost, and a lost chunk in these j chunks remains lost. The states of the remaining $(i-j)$ chunks do not transition.

In RAFI, a chunk failure is identified by independent identification thresholds, which is quite different from the traditional single identification threshold described in Section 1. For example, in a (6,3)-coded data center, stripe A has one failed chunk and is a low risk stripe, stripe B has two failed chunks and is a high risk stripe. A time threshold T_1 which is larger than the original time threshold T is set to identify failures of chunks in the low risk stripe; while a time threshold T_2, which is shorter than the T is set to identify failures of chunks in the high risk stripe. As shown in Figure 1, using RAFI, the failure identification of chunk a1 in the stripe A is postponed; in the meanwhile, the failure identification of chunks b1 and b2 in the stripe B is expedited.

RAFI is flexible. First, all the time thresholds can be set independently to get proper trade-offs between the data reliability and availability, and the repair network traffic for a certain type of stripes. Second, the identification thresholds can be merged to get proper trade-offs between the RAS and the computation cost of RAFI. When the time thresholds in all identification thresholds are set to the same T, RAFI becomes TFI.

3.2 Benefit and Cost

Improving the RAS: Using RAFI, we can independently set different time thresholds to identify failures. First, short thresholds are used to expedite the identification of failed chunks in high risk stripes, thus improving the data reliability and availability. At the same time, long thresholds are used to postpone the failure identification of chunks in low risk stripes, thus reducing the repair network traffic and improve the serviceability. Because the identification time is dominant in the repair time of chunks in high risk stripes, the expedition is effective in improving the data reliability and availability thus compensates the negative impacts induced by the postponement. Because most repair network traffic is induced by recovering chunks in low risk stripes, the repair network traffic is significantly reduced, even under the consideration of the extra repair network traffic induced by the expedition, thus improving the serviceability.

Compatibility: Because RAFI focuses on the failure identification phase, it can work together with existing optimizations which focus on the failure recovery phase.

Increasing Degraded Reads: Degraded read is an operation to read unavailable but recoverable chunks in a stripe. Because we postpone the failure identification of chunks in low risk stripes, more failed chunks might be generated, thus increasing degraded reads. However, the simulation results in Section 5 show that degraded reads increase by less than 1.7% due to RAFI. Because degraded reads are much fewer than normal reads, the overhead is very small.

4 Prototyping Evaluation

In this section, we first present the evaluation methodology; then we illustrate implementation details of our prototype RAFI-HDFS; finally we demonstrate results of prototyping experiments.

4.1 Evaluation Methodology

To verify the effectiveness of RAFI, we propose a hybrid methodology to comprehensively evaluate RAFI via both simulation and prototype implementation. The reason is explained below.

It is difficult to evaluate a technique targeting at the RAS of data centers because the data loss and unavailability events are very rare and not evenly distributed. The accuracy problem induced by that uneven distribution can be mitigated by high accurate simulation, which is run thousands to millions of iterations, although the simulator itself might be not that accurate. However, pure simulation cannot verify the correctness of design details and might cover fatal defects of the technique.

In our hybrid evaluation, the design details and computational cost of RAFI are verified through prototyping running on a real distributed storage system; the effectiveness and efficiency of RAFI on the RAS are evaluated through high accurate Monte-Carlo simulation.

In this section, we evaluate the identification time and computational cost of RAFI in real world environments. The simulator and simulation results are discussed in Section 5.

4.2 RAFI-HDFS

To evaluate the effectiveness of RAFI, we implement a prototype named RAFI-HDFS on HDFS [27], a representative distributed file system widely deployed in the data centers. Because erasure coding is supported by HDFS in version 3.0.0, our implementation is based on HDFS 3.0.0-alpha2. The implementation of RAFI-HDFS follows the design in Section 3. We only add about 200 lines of codes to HDFS.

Figure 2 demonstrates the overall architecture of RAFI-HDFS consisting of two modules: one is a classification module and the other is an identification module.

The classification module is responsible for converting the node failures into appropriate input for the identification thresholds. More specifically, the classification module receives a node list that contains all failed nodes and their failure durations from the node monitor module. Only those nodes whose failure durations are larger

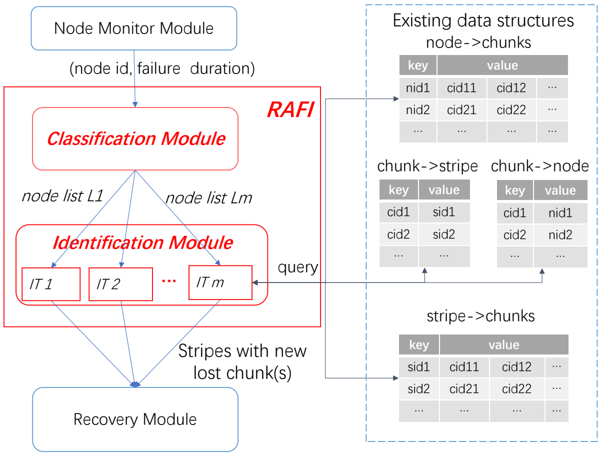


Figure 2: Architecture of RAFI-HDFS. The right side is existing data structures which are used in RAFI. The node monitor module reports failed nodes and their failure durations. The classification module inserts nodes to different identification thresholds in the identification module according to their failure durations. The identification thresholds (IT) in the identification module are used to identify chunk failures.

than T_i ($1 \leq i \leq m$) are inserted into the node hash list L_i for the *identification threshold (IT) i*, thus reducing the computation cost of that identification threshold. It is worth noting that the classification module replaces failed chunk lists with failed node lists. In such a manner, the memory usage of maintaining the numerous failed chunks is saved.

The identification module is a universal set of all the identification thresholds in RAFI. When *IT i* receives its node list L_i, it begins to calculate the *count* of failed chunks in stripes. First, the identification threshold calculates the *count* of unidentified failed chunks in stripes through querying the node-chunk mapping table and the chunk-stripe mapping table, which typically reside in the main memory of the manager nodes of the data centers. Second, through querying the stripe-chunk mapping table and chunk-node mapping table, the *count* of lost chunks is obtained. If the count of failed chunks (unidentified failed chunks and lost chunks) is larger than or equal to i, those failed chunks which belong to nodes in L_i, transition to lost.

After working through all identification thresholds, if new chunk failures are identified, the recovery module is called to recover stripes containing those lost chunks. Particularly, for nodes which enter *IT* 1, the failures of these nodes are identified and these nodes are removed from the system at the end of the *IT* 1.

Complexity. RAFI-HDFS only checks chunks on failed nodes which newly enter L_i to reduce the computation cost. Assume there are j nodes in L_i ($2 \leq i \leq m$) and there are an average of d chunks to be checked on the node. Each stripe has $k+m$ chunks. Because we use a hash list to track the failed nodes, the total comparison time is about $(k+m)\times d$. The time complexity of identifying chunk failures is $\mathcal{O}(d)$.

4.3 Results of Prototyping Experiments

Experimental Setups. The experimental system consists of ninety-seven servers running on the Alibaba Cloud [28]. One server served as a NameNode contains an Intel Xeon E5-2682v4 @ 2.5 GHZ CPU (4 vCPU), 16 GB DDR4 memory, 1.5 Gbps network and 40 GB SSD. The remaining 96 servers are used as DataNodes, each of which has an Intel Xeon E5-2680v3 @ 2.5 GHZ CPU (1 vCPU), 1 GB DDR4 memory, 1 Gbps network and 40 GB SSD. The operating system running on all these servers is Ubuntu 14.04. Each DataNode sends heartbeats to the NameNode every 3 seconds and the NameNode checks the states of all DataNodes every 5 minutes. As default in HDFS, the time threshold $T=10.5$ minutes and the erasure coding scheme RS(6,3) is used.

Identification Time of Chunks: The identification time of a chunk is the period from the time when a chunk becomes failed to the time when the chunk is identified as a lost chunk. In order to evaluate the real identification time, we collect the identification times by randomly killing two DataNodes. In order to evaluate the real identification time of chunks, we collect the identification times by randomly killing DataNodes in 0, 5, 10, and 20 minutes. Each experiment is conducted 20 times. In RAFI, T_2 is set to 1 minute and T_1 is set to 60 minutes. The results are consistent with our analysis in Section 3.2. The results demonstrate that TI_2 is expedited and TI_1 is postponed. When we simultaneously kill two storage nodes, TI_1 and TI_2 under TFI are 13.1 minutes; however, TI_2 under RAFI is 3.6 minutes, while TI_1 under RAFI is 62.6 minutes. Moreover, TI_1 and TI_2 are not relevant to the time between the failure arrivals.

Burden on the NameNode: Because the computations run on the NameNode, we record the time spent to identify chunk failures when nodes fail to further estimate the impact on the NameNode. The chunk size is shrunk to $1KB$ in our cluster to generate enough number of chunks. In the experiments, each DataNode stores about 68,000 chunks. In the experiments, there is no I/O workloads because the time spent to identify chunk failures under no I/O workloads is sufficient to indicate the overhead caused by RAFI on the NameNode. For each result, we concurrently kill DataNodes. Each experiment is conducted 10 times and we calculate the average results.

We evaluate the time spent to identify chunk failures from two aspects: the number of DataNodes in the cluster and the number of concurrent node failures.

First, as shown in Figure 3, the time spent to identify all 68,000 chunk failures on one failed DataNode increases from 74 to 137 milliseconds when the number of DataNodes increases from 12 to 96. Compared to time thresholds and check intervals (by default 10.5 and 5 minutes, respectively), the time spent to identify chunk failures can be negligible in the identification time.

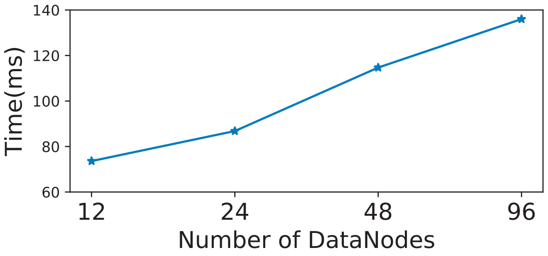


Figure 3: Time spent to identify chunk failures when when a DataNode fails. The number of DataNodes in the cluster changes from 12 to 96. E.g., the NameNode takes 87 ms to identify 68,000 chunk failures in a cluster of 24 DataNodes.

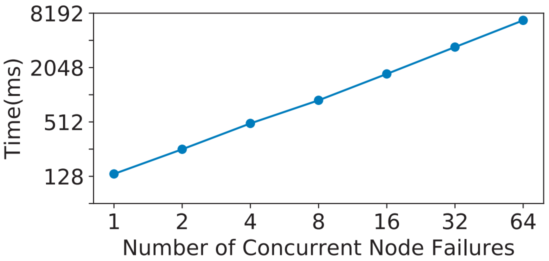


Figure 4: Time spent to identify chunk failures when DataNodes fail. The cluster consists of 96 DataNodes. E.g., the NameNode spends 889 ms to identify 544,000 chunk failures when eight DataNodes fail concurrently .

Second, as illustrated in Figure 4, the time spent to identify chunk failures increases linearly as concurrent node failures increase. The experiment results are consistent with the analysis in Section 4.2. It is worth noting that there are no failed nodes in most check time. Thus, our method has minimal impact on the NameNode.

Moreover, in our evaluation, only one single thread is used to check all chunks on failed nodes. In fact, we can use multi-threading technologies to check all chunks on failed nodes, e.g., each thread is responsible for checking all chunks on one failed node. Therefore, the time spent to identify all chunks on failed nodes can be dramatically reduced when multiple nodes fail concurrently.

5 Simulations and Results Analysis

In this section, we discuss our simulator and simulation results to evaluate the effectiveness and efficiency of RAFI on the RAS.

5.1 DR-SIM

We developed a simulator called DR-SIM which is written in the R language because it easily runs in parallel. The source code is approximately 1400 lines [29].

Event-driven simulators are widely used to study the RAS of data centers [14, 20, 30]. However, those simulators cannot be used in our simulations due to the following two reasons. First, some simulators are not open source, e.g., the Google's Cell Simulator [20]. Second, the RAS cannot be all simulated by some simulators. For example, limited by performance, the data reliability cannot be studied by the ds-sim [14]. As a result, we develop our own simulator, named DR-SIM, to study the effect of the data repair on the RAS in data centers.

We summarize important features of DR-SIM as follows. (1) The trade-off between the performance and accuracy of DR-SIM is carefully tuned. A simulation iteration (typically represents five years) can be finished in tens of seconds. Therefore, we run hundreds of thousands iterations for each simulation configuration, to accurately measure the RAS. (2) Many state-of-the-art optimizations on the data repair are integrated into DR-SIM, and important parameters of the data repair are considered as variants in DR-SIM. Through modifying the configuration of DR-SIM, we study the effectiveness and efficiency of RAFI upon various combinations of the state-of-the-art optimizations under various typical environments of the data centers.

Figure 5 shows the architecture of DR-SIM which includes four modules: a configuration manager, a failure generator, a repair calculator, and an event collector.

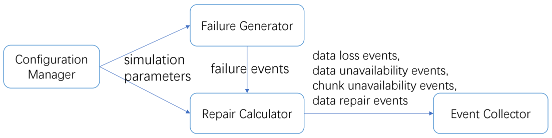


Figure 5: Architecture of DR-SIM

The configuration manager loads parameters used in the simulations. The parameters are explained as follows. (1) *System parameters*: The target erasure-coded data center consists of N independent storage nodes. Each node has d chunks. The chunk size is s. (2) *Coding parameters*: Data are coded with (k, m) erasure codes, i.e., k data chunks and m parity chunks are in a stripe. The $k+m$ chunks in the same stripe are distributed to $k+m$ distinct nodes. A random placement policy is used because it is usually adopted in practice. The recovery penalty factor of S^i ($1 \le i \le m$) is r_i which is between the amount data transmitted for recovery of S^i and s. The recovery network bandwidth is b on each node. (3) *Failure parameters*: Assume node failure arrivals are independent. Let $f(x)$ and $F(x)$ be the probability and cumulative distribution functions of the failure arrivals, respectively. Assume failure durations are independent. Let $g(x)$ and $G(x)$ be the probability and cumulative distribution functions of the failure durations, respectively. ρ is the ratio of permanent node failures to all node failures. τ denotes the additional proportion of correlated node failures. (4) *Identification parameters*: Storage nodes periodically send heartbeats to dedicated manager nodes, e.g. the NameNode [27, 31] or the metadata manager [32]. The manager nodes check states of all nodes at regular time intervals of T_h. The time thresholds for identifying chunk failures are T_i ($1 \le i \le m$). (5) *Simulation runtime parameters*: N_i denotes the number of iterations. T_d is the simulation duration for each iteration.

The failure generator is responsible for generating failure arrivals and failure durations of node failures at the beginning of a simulation iteration. The failure arrivals are generated according to the distribution function $f(x)$. Permanent failures and transient failures are generated by their durations. For the transient failures, their durations are generated according to the distribution function $g(x)$. For permanent failures, they are generated according to the parameter ρ. Technically, failure durations of the permanent failures are set to zero (only for handling but not calculating). In DR-SIM, additional correlated failures are explicitly generated by adding a random value between 0 to 120 seconds [20] to existing failure arrivals according to the parameter τ. It is worth noting that the comeback of transient failed nodes has been already considered in DR-SIM.

The repair calculator simulates the data repair process for lost chunks when failures occur. The repair calculator identifies the chunk failures according to the T_h and T_i ($1 \le i \le m$) and calculates the repair time for lost chunks based on the recovery network bandwidth, the recovery penalty factors and the recovery priority. The recovery processes of lost chunks are scheduled depending on the number of lost chunks in their stripes. For stripes have the same number of lost chunks, the repair calculator uses first come first scheduled rule to manage the recovery of those chunks. Moreover, lost chunks are recovered in parallel by utilizing all available nodes [33, 34].

The event collector is responsible for collecting data loss events, data unavailability events, chunk unavailability events, and data repair events. At the end of each iteration, DR-SIM calculates metrics according to the collected events. The mean time to data loss in the whole data center (referred as ***MTTDL***) is the metric to evaluate the data reliability. All the data loss events are recorded to calculate the *MTTDL*. The cumulative unavailable time of all stripes (referred as T_{us}) is the metric to evaluate the data availability. All the data unavailability events are recorded to calculate the T_{us}. The total repair network traffic (referred as ***RNT***) is the metric to evaluate the serviceability. All the data repair events are recorded to calculate the *RNT*. The cumulative unavailable time of all chunks (referred as T_{uc}) is the metric to evaluate the degraded reads. All the chunk unavailability events are recorded to calculate the T_{uc}. The former three metrics are widely used in evaluation of the RAS in the data centers [6, 7, 12, 14, 15, 20, 30, 35, 36]. The latter one is roughly in proportion to the number of degraded reads. It is worth noting that chunks and stripes are actually not simulated in DR-SIM under the consideration of computation complexity. In fact, the cumulative unavailability time of stripes and cumulative unavailability time of chunks are estimated from the generated node failures and data repair events.

5.2 Simulation Testbed

Comparisons between RAFI and TFI are made upon the testbed described as follows.

The following three state-of-the-art optimizations are always considered in the testbed. (1) The network adopts CLOS topologies [24–26]. (2) All lost chunks are parallel recovered via using available recovery network bandwidth on all nodes. (3) The chunks in stripes with more lost chunks have the higher priority to be recovered.

Three kinds of erasure codes are chosen in the simulations to understand the sensitivity to different erasure codes. RS codes are are a set of popular erasure codes which are widely used in real world distributed storage systems [12, 20, 21]. Zigzag codes [10] represent MDS (Maximum Distance Separable) codes with optimal recovery penalty factors. LRC codes [7] are representative non-MDS codes deployed in Windows Azure Storage.

The 1 Gbps network is chosen as the baseline in the testbed under the consideration of the cost-effectiveness in the erasure-coded data center, although we have found that RAFI is more efficient in reducing the *RNT* under the 40 Gbps network during studying the sensitivity of RAFI to the recovery network bandwidth.

Because chunks in low risk stripes are the optimization objects of both RAFI and *Lazy* [14], Lazy is considered in the testbed when we made dedicated comparisons between these two techniques in Section 5.3.4.

Default values of most parameters used in the simulations are listed in Table 2. The failure arrivals are assumed to be independent and exponentially distributed with the mean time to failure (MTTF = 7.1 days) [12,20]. The failure durations are assumed to be independent and Weibull distributed. We get sample values from [20] and model the failure durations with Weibull(113 seconds, 0.54), which is shown in Figure 6. The model fits well starting from 0.5 minutes.

Table 2: Symbols and Their Definitions

Symbol	Definition	Default Value
N	# of storage nodes in a data center	1000
d	# of chunks on a node	125,000
s	Chunk size	128 MB
T_h	Check interval of node states	5 minutes
b	Recovery network bandwidth on each node	0.1 Gbps
T_d	Duration of each iterations	5 years
N_i	# of iterations	500,000

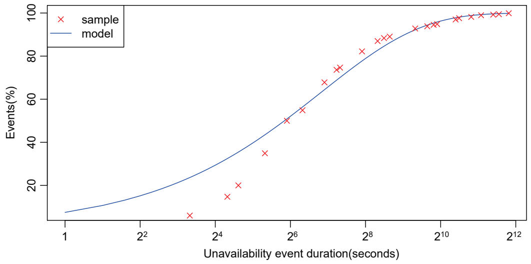


Figure 6: Unavailability Event Duration

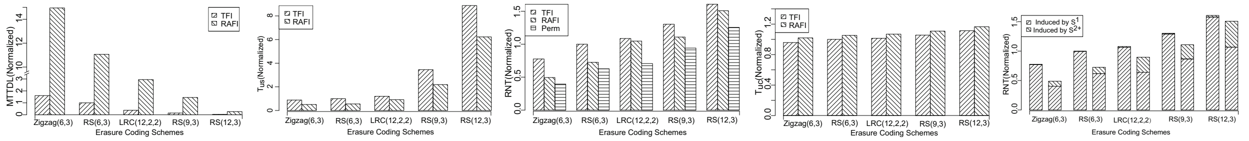
In our simulations, to simplify the comparison complexity, the identification thresholds *identification threshold i* ($i > 1$) are merged to one by sharing the same threshold value. The features of the erasure codes, and two time threshold values (one for T_1, and the other for T_i ($i > 1$)) are represented by an abbreviation, e.g., RS(6,3)-15-2 denotes a data center employed RS(6,3) with T_1 = 15 minutes and T_2 = T_3 = 2 minutes. r_1, r_2 and r_3 of an RS(6,3)-coded stripe are 6, 7, and 8, respectively. All measured metrics including the *MTTDL*, T_{us}, *RNT* and T_{uc}, are normalized to that of the RS(6,3)-15-15 (it denotes a TFI configuration when the latter two values are the same). The *MTTDL*, T_{us}, and *RNT* are the metrics to evaluate the RAS.

5.3 Simulation Results

5.3.1 RAS as Functions of T_i

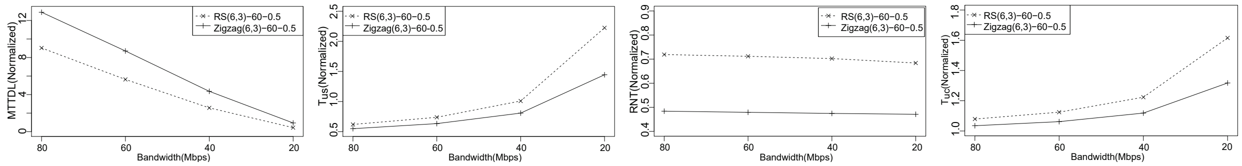
First of all, we run simulations to find the proper two threshold values for RAFI. Let $T_3 = T_2 = T_1$. Figure 9a illustrates that the data reliability and availability get worse while the repair network traffic is improved when T_1 increase. The *RNT* reduces slowly when T_1 is larger than 60 minutes. Thus, T_1 of RAFI is set to 60 minutes in the rest simulations.

Then, to study the impact of T_2, let $T_3 = T_2$. T_2 ranges from 0.5 to 8 minutes. The results in Figure 9b demonstrate that RAFI simultaneously improves the RAS in most configurations. More specifically, the *MTTDL* is improved by a factor up to 11. The T_{us} is reduced by up to 45%. The *RNT* is reduced by up to 27%. The *RNT* increases with the reduction of T_2 because reducing T_2 increases the number of S^{2+}, and results in unnecessary



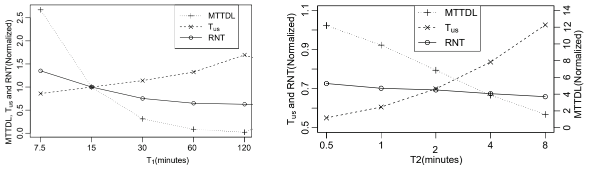
(a) Reliability (b) Availability (c) Serviceability (d) Degraded Reads (e) *RNT* induced by S^1 v.s. *RNT* induced by S^{2+}.

Figure 7: Impacts of different erasure coding schemes on the RAS. The results are normalized to RS(6,3)-15-15.



(a) Reliability (b) Availability (c) Serviceability (d) Degraded Reads

Figure 8: Impacts of constrained recovery network bandwidth on the RAS. RS(6,3) and Zigzag(6,3) are considered in the simulations. The results are normalized to RS(6,3)-15-15.



(a) The *MTTDL*, T_{us}, and *RNT* with different T_1.

(b) The *MTTDL*, T_{us}, and *RNT* with different T_2. T_1 is set to 60 minutes.

Figure 9: Impacts of T_1 and T_2. The erasure coding scheme is RS(6,3), and the results are normalized to RS(6,3)-15-15.

repair network traffic to repair those S^{2+}. Only when T_2 is 8 minutes, which is close to the original T of 15 minutes, RAFI does not take effect on the data availability.

From the results, we find that the data reliability and availability are sensitive to the decrease of T_2 but the repair network traffic is not sensitive to the decrease of T_2. As a result, *both* T_2 *and* T_3 *are set to 0.5 minutes* in the rest simulations.

5.3.2 RAS as Functions of Erasure Coding Schemes

In this section, we examine the effectiveness and efficiency of RAFI under five typical kinds of erasure coding schemes, RS(6,3), RS(9,3), RS(12,3), Zigzag(6,3) [10], and LRC(12,2,2) [7]. These erasure coding schemes represent various recovery penalty factors. T_1, T_2 and T_3 are 60 minutes, 0.5 minutes and 0.5 minutes, respectively. All results are normalized to RS(6,3)-15-15 and presented in Figure 7. In general, RAFI can cooperate with all the five kinds of erasure coding schemes, and simultaneously further improve the RAS at the cost of the slightly increased degraded reads.

Improving Reliability: Figure 7a shows that RAFI improves the *MTTDL* of Zigzag(6,3), RS(6,3), LRC(12,2,2), RS(9,3), and RS(12,3) by a factor of 9.3, 11, 7.7, 9.8, and 7.7, respectively. When the recovery penalty factor increases, the improvements diminish a little. The reason is that the higher recovery penalty factor lengthens the recovery time, thus weakens the effect of the reduction of the identification time.

Improving Availability: Figure 7b illustrates that RAFI improves the data availability under various erasure coding schemes. The T_{us} of Zigzag(6,3), RS(6,3), LRC(12,2,2), RS(9,3), and RS(12,3) is reduced by 43%, 45%, 24%,37%, and 30%, respectively.

Improving Serviceability: Figure 7c shows that RAFI reduces the *RNT* under various erasure coding schemes. The *Perm* represents the *RNT* induced only by permanent node failures. Figure 7e shows the composition of the *RNT*. In TFI, over 99% of the *RNT* is induced by the repair of S^1. In RAFI, about 15%-30% of the *RNT* is induced by the repair of S^{2+}.

Degraded Reads: When RAFI postpones the recovery of S^1, the amount of unidentified failed chunks increases. Figure 7d shows that the degraded reads increase by 1.7% at most, which is very slight.

5.3.3 RAS as Functions of Recovery Network Bandwidth

Network bandwidth is very valuable in the data centers. In this section, simulations are performed to understand the effect of RAFI under a limited recovery network bandwidth b. Both RS(6,3) and Zigzag(6,3) codes are considered in the simulations. T_1, T_2 and T_3 are 60 minutes, 0.5 minutes and 0.5 minutes, respectively. The simulation results are normalized to RS(6,3)-15-15 and presented in Figure 8.

Figure 8 shows that the RAS are still improved even when b is 40 Mbps. However, at the same time, the T_{uc} increases by 22%, because a small b significantly extends the repair time of the lost chunks, thus leads to longer chunk unavailability time. When b reduces, the reduction of *RNT* increases a little.

Table 3: The RAS improvements under 40 Gbps network

Erasure Coding Schemes	RS(6,3)	Zigzag(6,3)
Improvement of *MTTDL*	3.4	3.7
Reduction of T_{us}	54%	56%
Reduction of *RNT*	79%	86%

40 Gbps network: Nowadays, some data centers are equipped with 40 Gbps network for each node [26, 37]. In such a scenario, the recovery network bandwidth b is 4 Gbps for each node. Table 3 shows that RAFI still improves the RAS when b is 4 Gbps. When b increases from 100 Mbps to 4 Gbps, the recovery time reduces. Because the ratio between the recovery time and the repair time decreases, the improvement of *MTTDL* decreases. However, when the repair rate increases, there will be more unnecessary repair network traffic. Therefore, RAFI is very effective in reducing the repair network traffic.

5.3.4 Comparisons with *Lazy*

To comprehensively compare RAFI with *Lazy*, the comparisons are made in the form of TFI + *Lazy* v.s. RAFI + *Lazy* v.s. RAFI. RS(6,3) and Zigzag(6,3) codes are considered in the simulations. *Lazy* [14] recovers lost chunks if their host stripes have at least two lost chunks. In TFI + *Lazy*, we use the parameters: T_1 = T_2 = T_3 = 15 minutes. In RAFI + *Lazy*, T_1 = T_2 = 15 minutes, T_3 = 1 minutes. In RAFI, T_1 = 60 minutes and T_2 = T_3 = 15 minutes. The comparison results are shown in Figure 10.

Cooperating with *Lazy*, compared to TFI, RAFI improves the *MTTDL* by a factor of 5.1, at the cost of increasing the *RNT* by 2.5%. Because *Lazy* even does not recover some permanent failed chunks, RAFI cannot further reduce the *RNT*.

Compared to TFI + *Lazy*, RAFI without *Lazy* increases the *MTTDL* by over two orders of magnitude at a higher *RNT* cost. An interesting thing is that, RAFI suffers a much lower increase of the *RNT* when cooperating with the Zigzag codes. The reason is that the recovery penalty factor of a Zigzag(6,3)-coded S^1 is only 63% of that of an RS(6,3)-coded S^1. In fact, as mentioned in Section 6, many codes [6, 7] are proposed to reduce the recovery penalty factor of stripes with one lost chunk.

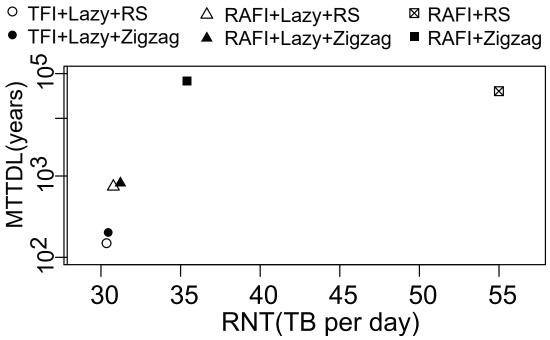


Figure 10: The *MTTDL* and *RNT* under TFI+*Lazy*, RAFI+*Lazy*, and RAFI. The erasure coding schemes are RS(6,3) and Zigzag(6,3). X axis is the repair network traffic, Y axis is the *MTTDL*.

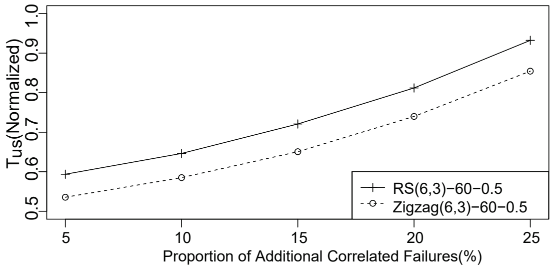


Figure 11: Impact of correlated failures on availability. The results are normalized to RS(6,3) with no additional correlated failures.

5.3.5 Availability under Correlated Failures

Because transient failures may happen concurrently [20], we desire to see how data availability is affected by correlated failures. From Figure 11, we can see that, as the proportion of additional correlated failures increases, RAFI still reduces about 40% of the T_{us}, demonstrating that RAFI is very resilient to correlated failures.

6 Related Work

Existing solutions which are proposed to improve the RAS focus on optimizing the failure recovery phase, such as reducing recovery penalty factors [2, 4, 5, 7–13, 16, 17, 22, 23], improving recovery rates [6, 18, 19], and risk-aware recovery scheduling [3, 7, 14].

Reducing recovery penalty factors: Both the recovery time and repair network traffic are improved through reducing the recovery penalty factors of erasure codes. Two types of techniques are proposed. One is to design MDS and non-MDS erasure codes with low recovery penalty factors [2, 6–12, 15–17, 38]. The other is to design recovery algorithms to reduce recovery penalty factors of existing erasure codes [4, 5, 13].

Regenerating Codes [22, 23, 38, 39] are a family of MDS codes. The recovery penalty factors of the Regenerating Codes are much lower than that of the traditional RS (Reed-Solomon) codes [40]. However, the Regenerating Codes are not systematic codes, thus suffer from high read cost. To maintain low recovery penalty factors and read cost, systematic MDS codes, such as Zigzag and Butterfly codes [10, 17] are proposed. Zigzag codes [10] are proved to be with optimal recovery penalty factors in all systematic MDS codes. One significant drawback of Zigzag codes is that the implementation depends on non-binary algebra.

New trade-off points between storage overheads and recovery penalty factors are found through non-MDS codes, such as LRC [7, 11, 16]. Compared to MDS codes, non-MDS codes dramatically reduce the recovery penalty factors. However, the cost of non-MDS codes cannot be ignored, particular when the scale of the data center is very large, i.e., even 1% extra storage overhead usually means millions of dollars [41, 42].

Recovery algorithms, such as [4,5,13], are proposed to reduce recovery penalty factors of existing erasure codes. The biggest drawback of those recovery algorithms is that their efficiency on reducing recovery penalty factors are much lower than that of designing novel codes.

Improving the recovery rate: Another approach to shorten the recovery time is improving the recovery rate.

It is common to improve the recovery rate through deploying high-speed networks, i.e., increasing the recovery network bandwidth. For example, CLOS networks [24–26] are deployed in FDS [6] to provide non-oversubscribed full bisection bandwidth networks at the scale of a data center. As a result, the recovery is dramatically accelerated.

The recovery rate is also improved through increasing the recovery parallelism. Mitra et al. propose a parallel chunk recovery method PPR [18] to improve the recovery parallelism. Li et al. propose a pipelined chunk recovery method ECPipe [19] to further improve that recovery parallelism. However, both PPR and ECPipe take effect when there are only a few chunks be recovered.

Risk-aware recovery scheduling: Besides accelerating the recovery of all chunks, high data reliability and availability can also be achieved through scheduling the recovery of chunks according to the *number of lost chunks* in their host stripes, which indicates the data reliability and availability risk of those stripes.

The recovery of the chunks in high risk stripes is prioritized in HDFS [3] and WAS [7]. In such a manner, the repair time of high risk stripes is dramatically reduced. Meanwhile, the increase of the repair time is relatively small. Therefore, the data reliability and availability are improved. It is worth noting that, after being scheduled, the failure identification time becomes dominant in the repair time of high risk stripes, because those chunks in high risk stripes are usually very few. **As a result, the reduction in the identification time of high risk stripes is very effective in improving the data reliability and availability.**

Silberstein et al. propose a technique *Lazy* [14] to reduce the repair network traffic. Because chunks in low risk stripes, e.g., S^1, are dominant in all chunks be recovered, most of the repair network traffic is generated by recovering those chunks. Canceling the recovery of chunks in low risk stripes dramatically reduces the repair network traffic. However, the data reliability and availability dramatically decrease.

7 Conclusions

In this paper, we present a risk-aware failure identification scheme, named RAFI, to simultaneously improve the data reliability, availability, and serviceability (RAS) of erasure-coded data centers. The basic idea of RAFI is identifying a chunk failure not only according to its failure duration, but also based on the data reliability and availability of its host stripe. The benefits of RAFI are: (1) the identification of failed chunks in high risk stripes is expedited to improve the data reliability and availability; and (2) the identification of failed chunks in low risk stripes is postponed to reduce the repair network traffic, thus improving the serviceability. Our results based on both simulations and prototyping have demonstrated the effectiveness and efficiency of RAFI in terms of reduced data loss, unavailability, and repair network traffic.

8 Acknowledgment

We are grateful to anonymous reviewers and particularly our shepherd Dahlia Malkhi for their helpful comments and suggestions. We thank Changsheng Xie and Ting Ye for their great efforts on this paper. Shenggang Wan is the corresponding author. This research is sponsored by National Natural Science Foundation of China Grants Nos. 61300046 and 61331010, and U.S. National Science Foundation Grants Nos. CCF-1717660, CNS-1702474 and CCF-1547804.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System,” in *SOSP’03*, 2003.

[2] K. M. Greenan, X. Li, and J. J. Wylie, “Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs,” in *MSST’10*, 2010.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop Distributed File System,” in *MSST’10*, 2010.

[4] L. Xiang, Y. Xu, J. C. S. Lui, and Q. Chang, “Optimal recovery of single disk failure in RDP code storage systems,” in *SIGMETRICS’10*, 2010.

[5] S. Li, Q. Cao, J. Huang, S. Wan, and C. Xie, “PDRS: A New Recovery Scheme Application for Vertical RAID-6 Code,” in *NAS’11*, 2011.

[6] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue, “Flat Datacenter Storage,” in *OSDI’12*. USENIX, 2012.

[7] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, “Erasure Coding in Windows Azure Storage,” in *ATC’12*, 2012.

[8] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang, “Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads,” in *FAST’12*, 2012.

[9] Y. Hu, H. C. H. Chen, P. P. C. Lee, and Y. Tang, “NCCloud: applying network coding for the storage repair in a cloud-of-clouds,” in *FAST’12*, 2012.

[10] I. Tamo, Z. Wang, and J. Bruck, “Zigzag Codes: MDS Array Codes With Optimal Rebuilding,” *Transactions on Information Theory*, 2013.

[11] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, “XORing Elephants: Novel Erasure Codes for Big Data,” in *VLDB’13*, 2013.

[12] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, “A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster,” in *HotStorage’13*, 2013.

[13] S. Xu, R. Li, P. P. C. Lee, Y. Zhu, L. Xiang, Y. Xu, and J. C. S. Lui, “Single Disk Failure Recovery for X-Code-Based Parallel Storage Systems,” *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 995–1007, 2014.

[14] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin, “Lazy Means Smart: Reducing Repair Bandwidth Costs in Erasure-coded Distributed Storage,” in *SYSTOR’14*, 2014.

[15] K. V. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran, “Having your cake and eating it too: jointly optimal erasure codes for I/O, storage and network-bandwidth,” in *FAST’15*, 2015.

[16] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, “A Tale of Two Erasure Codes in HDFS,” in *FAST’15*, 2015.

[17] L. Pamies-Juarez, F. Blagojević, R. Mateescu, C. Gyuot, E. E. Gad, and Z. Bandić, “Opening the Chrysalis: On the Real Repair Performance of MSR Codes,” in *FAST’16*, 2016.

[18] S. Mitra, R. Panta, M. R. Ra, and S. Bagchi, “Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage,” in *EUROSYS’16*, 2016.

[19] R. Li, X. Li, P. P. C. Lee, and Q. Huang, “Repair Pipelining for Erasure-Coded Storage,” in *ATC’17*, 2017.

[20] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, “Availability in Globally Distributed Storage Systems,” in *OSDI’10*, 2010.

[21] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, “The quantcast file system,” 2013.

[22] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, “Network coding for distributed storage systems,” *Transactions on Information Theory*, 2010.

[23] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, and C. Huang, “Simple regenerating codes: Network coding for cloud storage,” in *INFOCOM’12*, 2012.

[24] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in *SIGCOMM’08*, 2008.

[25] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, and M. Zhu, “B4: experience with a globally-deployed software defined wan,” in *SIGCOMM’13*, 2013.

[26] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, and P. Germano, “Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network,” in *SIGCOMM’15*, 2015.

[27] D. Borthakur, “The hadoop distributed file system: Architecture and design,” *Hadoop Project Website*, vol. 11, 2007.

[28] AlibabaCloud, “Alibab ECS,” https://www.alibabacloud.com/product/ecs, 2017.

[29] J. Fang, “DR-SIM,” https://github.com/yydfjt/distributed_system_simulator, 2017.

[30] L. Wan, F. Wang, H. S. Oral, S. S. Vazhkudai, and Q. Cao, *A Report on Simulation-Driven Reliability and Failure Analysis of Large-Scale Storage Systems*, Nov 2014. [Online]. Available: http://www.osti.gov/scitech/servlets/purl/1185665

[31] A. Oriani and I. C. Garcia, “From Backup to Hot Standby: High Availability for HDFS,” in *SRDS’12*, 2012.

[32] A. Thomson and D. J. Abadi, “CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems,” in *FAST’15*, 2015. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/thomson

[33] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, “Fast Crash Recovery in RAMCloud,” in *SOSP’11*, 2011.

[34] B. gon Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris, “Efficient replica maintenance for distributed storage systems,” in *NSDI’06*, 2006.

[35] V. Venkatesan, I. Iliadis, and R. Haas, “Reliability of Data Storage Systems under Network Rebuild Bandwidth Constraints,” in *MASCOTS’12*, 2012.

[36] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, “Surviving failures in bandwidth-constrained datacenters,” in *SIGCOMM’12*, 2012.

[37] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the Social Network’s (Datacenter) Network,” in *SIGCOMM’15*, 2015.

[38] F. André, A.-M. Kermarrec, E. Le Merrer, N. Le Scouarnec, G. Straub, and A. Van Kempen, “Archiving cold data in warehouses with clustered network coding,” in *EUROSYS’14*, 2014.

[39] S. Jiekak, A.-M. Kermarrec, N. Le Scouarnec, G. Straub, and A. Van Kempen, “Regenerating Codes: A System Perspective,” in *SIGOPS’13*, 2013.

[40] I. S. Reed and G. Solomon, “Polynomial Codes Over Certain Finite Fields,” *Journal of the Society for Industrial and Applied Mathematics*, 1960.

[41] A. K. Dutta and R. Hasan, “How much does storage really cost? Towards a full cost accounting model for data storage,” in *International Conference on Grid Economics and Business Models*. Springer, 2013.

[42] Amazon, “Pricing of Amazon S3,” https://aws.amazon.com/s3/pricing, 2017.

Siphon: Expediting Inter-Datacenter Coflows in Wide-Area Data Analytics

Shuhao Liu, Li Chen and Baochun Li
Department of Electrical and Computer Engineering, University of Toronto

Abstract

It is increasingly common that large volumes of production data originate from geographically distributed datacenters. Processing such datasets with existing data parallel frameworks may suffer from significant slowdowns due to the much lower availability of inter-datacenter bandwidth. Thus, it is critical to optimize the delivery of inter-datacenter traffic, especially *coflows* that imply application-level semantics, to improve the performance of such geo-distributed applications.

In this paper, we present *Siphon*, a building block integrated in existing data parallel frameworks (*e.g.,* Apache Spark) to expedite their generated inter-datacenter coflows at runtime. Specifically, Siphon serves as a transport service that accelerates and schedules the inter-datacenter traffic with the awareness of workload-level dependencies and performance, while being completely transparent to analytics applications. Novel intra-coflow and inter-coflow scheduling and routing strategies have been designed and implemented in Siphon, based on a software-defined networking architecture.

On our cloud-based testbeds, we have extensively evaluated Siphon's performance in accelerating coflows generated by a broad range of workloads. With a variety of Spark jobs, Siphon can reduce the completion time of a single coflow by up to 76%. With respect to the average coflow completion time, Siphon outperforms the state-of-the-art scheme by 10%.

1 Introduction

Big data analytics applications are typically developed with modern data parallel frameworks, such as Apache Hadoop [1] and Spark [25], taking advantage of their out-of-the-box features of scalability. With the trend of further scaling out, it has been reported that these applications are deployed across the globe, with their raw input data generated from different locations and stored in geographically distributed datacenters [19, 24]. When processing such geo-distributed data, computation tasks in different datacenters would transfer their intermediate results across the inter-datacenter network, which has much lower bandwidth, typically by an order of magnitude [12], than that within a datacenter. As such, applications that involve heavy inter-datacenter traffic easily suffer from significantly degraded performance, known as the *wide-area data analytics* [23].

To alleviate such performance degradation, existing work in the literature has largely focused on rearranging the pattern of inter-datacenter traffic, with the hope of relieving network bottlenecks. Specifically, one category of such efforts [19, 23, 24] attempted to design optimal mechanisms of assigning input data and computation tasks across datacenters, to reduce or balance the network loads. Another category of the existing work [12, 22] tried to adjust the application workloads towards reducing demands on inter-datacenter communications.

However, given particular traffic from an application, improving its performance by directly accelerating the completion of its inter-datacenter data transfers has been largely neglected. To fill this gap, we propose a deliberate design of a fast delivery service for data transfers across datacenters, with the goal of improving application performance from an orthogonal and complementary perspective to the existing efforts. Moreover, it has been observed that an application cannot proceed until all its flows complete [7], which indicates that its performance is determined by the collective behavior of all these flows, rather than any individual ones. We incorporate the awareness of such an important application semantic, abstracted as *coflows* [8], into our design, to better satisfy application requirements and further improve application-level performance.

Existing efforts have investigated the scheduling of coflows within a single datacenter [6, 8, 17, 27], where the network is assumed to be congestion free and abstracted as a giant switch. Unfortunately, such an assumption no longer holds in the wide area inter-datacenter network, yet the requirement for optimal coflow scheduling to improve application performance becomes even more critical.

In this paper, we present three inter-datacenter coflow scheduling strategies that can significantly improve application-level performance. *First*, we have designed a novel and practical inter-coflow scheduling algorithm to minimize the average coflow completion time, despite the unpredictable available bandwidth in wide-area networks. The algorithm is based on Monte Carlo simulations to handle the uncertainty, with several optimizations to ensure its timely completion and enforcement. *Second*, we have proposed a simple yet effective intra-coflow scheduling policy. It tries to prioritize a subset of flows such that the potential straggler tasks can be accelerated. *Finally*, we have designed a greedy multi-path routing algorithm, which detours a subset of the traffic on a bottlenecked link to an alternate idle path, such that the slowest flow in a shuffle can be finished earlier.

Further, to enforce these scheduling strategies, we have designed and implemented *Siphon*, a new building block for data parallel frameworks that is designed to provide a transparent and unified platform to expedite inter-datacenter coflows.

From the perspective of data parallel frameworks, Siphon decouples *inter-datacenter* transfers from *intra-datacenter* traffic, serving as a transport with full coflow awareness. It can be easily integrated to existing frameworks with minimal changes in source code, while being completely transparent to the analytics applications atop. We have integrated Siphon to Apache Spark [25].

With Siphon, the aforementioned coflow scheduling strategies become feasible thanks to its software-defined networking architecture. For the datapath, it employs *aggregator* daemons on all (or a subset of) workers, forming a virtual overlay network atop the inter-datacenter WAN, aggregating and forwarding inter-datacenter traffic efficiently. At the same time, a *controller* can make centralized routing and scheduling decisions on the aggregated traffic and enforce them on aggregators. Also, the controller can work closely with the resource scheduler of the data parallel framework, to maintain a global and up-to-date knowledge about ongoing inter-datacenter coflows at runtime.

We have evaluated our proposed coflow scheduling strategies with Siphon. Across five geographical regions on Google Cloud, we have evaluated the performance of Siphon from a variety of aspects, and the effectiveness of intra-coflow scheduling in accelerating several real Spark jobs. Our experimental results have shown an up to 76% reduction in the shuffle read time. Further experiments with the Facebook coflow benchmark [8] have shown an $\sim$ 10% reduction on the average coflow completion time as compared to the state-of-the-art schemes.

We make three original contributions in this paper:

- We have proposed a novel and practical inter-coflow scheduling algorithm for wide-area data analytics. Starting from analyzing the network model, new challenges in inter-datacenter coflow scheduling have been identified and addressed.
- We have designed an intra-coflow scheduling policy and a multi-path routing algorithm that improve WAN utilization in wide-area data analytics.
- We have built Siphon, a transparent and unified building block that can easily extend existing data parallel frameworks with out-of-box capability of expediting inter-datacenter coflows.

2 Motivation and Background

In modern big data analytics, the network stack traditionally serves to deliver *individual flows* in a timely fashion [3, 4, 26], while being oblivious to the application workload. Recent work argues that, by leveraging workload-level knowledge of flow interdependence, the proper scheduling of coflows can improve the performance of applications in datacenter networks [8].

As an application is deployed at an inter-datacenter scale, the network is more likely to be a system bottleneck [19]. Existing efforts in wide-area data analytics [12, 19, 22] all seek to avoid this bottleneck, rather than mitigating it. Therefore, it is necessary to enforce a systematic way of scheduling inter-datacenter coflows for better link utilization, given the fact that the timely completion of coflows can play an even more significant role in application performance.

However, new challenges arise in inter-datacenter networks, which have quite different characteristics as compared to datacenter networks [11, 14]. Such unique characteristics can invalidate the assumptions made by existing coflow scheduling algorithms.

First, inter-datacenter networks have a different network model. Networks are usually modeled as a big switch [8] or a fat tree [20] in the recent coflow scheduling literature, where the ingress and egress ports at the workers are identified as the bottleneck. This is no longer true in wide area data analytics, as the available bandwidth on inter-datacenter links are magnitudes lower than the edge capacity (see Table 1).

Second, the available inter-datacenter bandwidth fluctuates over time. Unlike in datacenter networks, the completion time of a given flow can hardly be predictable, which makes the effectiveness of existing deterministic scheduling strategies (*e.g.*, [8, 27]) questionable. The reason is easily understandable: though the aggregated link bandwidth between a pair of datacenters might be abundant, it is shared among tons of users and their

	Oregon	Carolina	Tokyo	Belgium	Taiwan
Oregon	**3000**	236	250	152.0	194
Carolina	237	**3000**	83.8	251	45.1
Tokyo	83.8	81.7	**3000**	89.2	586
Belgium	249	242	86.6	**3000**	76.0
Taiwan	182	35.8	508	68.0	**3000**

Table 1: Peak TCP throughput (Mbps) achieved across different regions on the Google Cloud Platform, measured with `iperf3` in TCP mode on standard 2-core instances. Rows and columns represent source and destination datacenters, respectively. These statistics match the reports in [12].

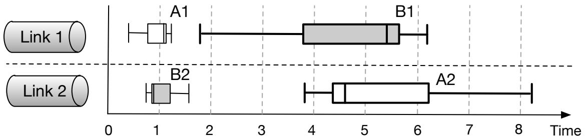


Figure 1: An example with two coflows, A and B, being sent through two inter-datacenter links. Based on link bandwidth measurements and flow sizes, the duration distributions of four flows are depicted with box plots. Note that the expected duration of $A1$ and $B2$ are the same.

launched applications, with varied, unsynchronized and unpredictable networking patterns.

Third, our ability to properly schedule and route inter-datacenter flows is limited. We may gain full control via software-defined networking within a datacenter [28], but such a technology is not readily available in inter-datacenter WANs. Flows through inter-datacenter links are typically delivered with best effort on direct paths, without the intervention of application developers.

To summarize, it calls for a redesigned coflow scheduling and routing strategy for wide-area data analytics, as well as a new platform to realize in existing data analytics frameworks. In this paper, Siphon is thus designed from the ground up for this purpose. It is an application-layer, pluggable building block that is readily deployable. It can support a better WAN transport mechanism and transparently enforce a flexible set of coflow scheduling disciplines, by closely interacting with the data parallel frameworks. A Spark job with tasks across multiple datacenters, for example, can take advantage of Siphon to improve its performance by reducing its inter-datacenter coflow completion times.

3 Scheduling Inter-Datacenter Coflows

3.1 Inter-Coflow Scheduling

Inter-coflow scheduling is the primary focus of the literature [6, 8, 21, 27]. In this section, we first analyze the practical network model of wide-area data analytics. Based on the new observations, we propose the details of a Monte Carlo simulation-based scheduling algorithm.

3.1.1 Goals and Non-Goals

Our major objective is to *minimize the average coflow completion time*, in alignment with the existing literature. However, we focus on inter-datacenter coflows, which are constrained by a different network model. In particular, based on the measurement in Table 1, we conclude that inter-datacenter links are the only bottlenecked resources, and congestion can hardly happen at the ingress or egress port. For convenience, we call it a *dumb bell* network structure. In addition, we consider the availability of inter-datacenter bandwidth as a dynamic resource. Scheduling across coflows should take runtime variations into account, making a scheduling decision that has a higher probability of completing coflows faster.

Similar to [8, 28], we assume the complete knowledge of ongoing coflows, *i.e.,* the source, the destination and the size of each flow are known as soon as the coflow arrives. Despite recent work [6, 27] which deals with zero or partial prior knowledge, we argue that this assumption is practical in modern data parallel frameworks. It is conceivable that the task scheduler is fully aware the potential cross-worker traffic before launching the tasks in the next stage and triggering the communication stage [1, 7, 25]. We will elaborate further on its feasibility in Sec. 4.3.

3.1.2 Schedule with Bandwidth Uncertainty

Coflow scheduling in a big switch network model has been proven to be NP-hard, as it can be reduced to an instance of the concurrent open shop scheduling with coupled resources problem [8]. With a dumb bell network structure, as contention is removed from the edge, each inter-datacenter link can be considered an independent resource that is used to service the coflows (jobs). Therefore, it makes sense to perform fully preemptive coflow scheduling, as resource sharing always results in an increased average [10].

The problem may seem simpler with this network model. However, it is the sharing nature of inter-datacenter links that complicates the scheduling. The real challenge is, being shared among tons of unknown users, the available bandwidth on a certain link is not predictable. In fact, the available bandwidth a random variable whose distribution can be inferred from history measurements. Thus, the flow durations are also random variables. The coflow scheduling problem in wide-area data analytics can be reduced to the *independent probabilistic job shop scheduling problem* [5], which is also NP-hard.

We seek a heuristic algorithm to solve this online scheduling problem. An intuitive approach is to make an estimation of the flow completion times, *e.g.,* based on the expectation of recent measurements, such that we can solve the problem by adopting a deterministic scheduling policy such as Minimum-Remaining Time First (MRTF) [8, 27].

Unfortunately, this naive approach fails to model the

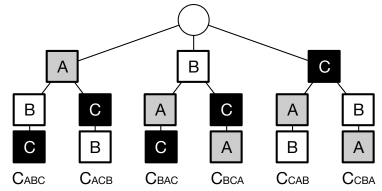


Figure 2: The complete execution graph of Monte Carlo simulation, given 3 ongoing coflows, A, B and C. The coflow scheduling order is determined by the distributions at the end of all branches.

probabilistic distribution of flow durations. Fig. 1 shows a simple example in which deterministic scheduling does not work. In this example, the available bandwidth on Link 1 and 2 have distinct distributions because users sharing the link have distinct networking behaviors. With Coflow A and B competing, the box plots depict the skewed distributions of flow durations if the corresponding coflow gets all the available bandwidth.

With a naive, deterministic approach that considers average only, scheduling either A or B will result in a minimum average coflow completion time. However, it is an easy observation that, with a higher probability, the duration of flow $A1$, will be shorter than $B2$. Thus, prioritizing Coflow A over B should yield an optimum schedule.

3.1.3 Monte Carlo Simulation-based Scheduling

To incorporate such uncertainty, we propose an online Monte Carlo simulation-based inter-coflow scheduling algorithm, which is greatly inspired by the offline algorithm proposed in [5].

The basic idea of Monte Carlo simulation is simple and intuitive: For every candidate scheduling order, we repeatedly simulate its execution and calculate its *cost*, *i.e.,* the simulated average coflow completion time. With enough rounds of simulations, the cost distribution will approximate the actual distribution of average coflow completion time. Based on this simulated cost distribution, we can choose among all candidate scheduling orders at a certain confidence level.

As an example, Fig. 2 illustrates an algorithm execution graph with 3 ongoing coflows. There are 6 potential scheduling orders, corresponding to the 6 branches in the graph. To perform one round of simulation, the scheduler generates a flow duration for each of the node in the graph, by randomly drawing from their estimated distributions. By summing up the cost for each branch, it yields a best scheduling decision instance, which results in a counter increment. After plenty of rounds, the best scheduling order will converge to the branch with the maximum counter value.

One major concern of this algorithm is its high complexity. With n ongoing coflows, there will be up to $n!$ branches in the graph of simulation. Luckily, thanks to the nature of coflow scheduling, we can apply the following techniques to limit the simulation search space.

Bounded search depth. In online coflow scheduling, all we care about is *the* coflow that should be scheduled *next*. This property makes a full simulation towards all leaf nodes unnecessary. Therefore, we set an upper bound, d, to the search depth, and simulate the rest of branches using MRTF heuristic and the median flow durations. This way, the search space is limited to a polynomial time $\Theta(n^d)$.

Early termination. Some "bad" scheduling decisions can be identified easily. For example, scheduling an elephant coflow first will always result in a longer average. Based on this observation, after several rounds of full simulation, we cut down some branches where performances are always significantly worse. This technique limit the search breath, resulting in a $O(n^d)$ complexity.

Online incremental simulation. As an online simulation, the scheduling algorithm should quickly react to recent events, such as coflow arrivals and completions. Whenever a new event comes, the previous job execution graph will be updated accordingly, by pruning or creating branches. Luckily, the existing useful simulation results (or partial results) can be preserved to avoid repetitive computation.

These optimizations are inspired by similar techniques adopted in Monte Carlo Tree Search (MCTS), but our algorithm differs from MCTS conceptually. In every simulation, MCTS tends to reach the leave of a single branch in the decision tree, where the outcome can be revealed. As a comparison, our algorithm has to go though all branches at a certain depth, otherwise we cannot figure out the optimal scheduling for the particular instance of available bandwidth.

3.1.4 Scalability

In wide-area data analytics, a centralized Monte Carlo simulation-based scheduling algorithm may be questioned with respect to its scalability, as making and enforcing a scheduling decision may experience seconds of delays.

We can exploit the parallelism and staleness tolerance of our algorithm. The beauty of Monte Carlo simulation is that, by nature, the algorithm is infinitely parallelizable and completely agnostic to staled synchronization. Thus, we can potentially scale out the implementation to a great number of scheduler instances placed in all worker datacenters, to minimize the running time of the scheduling algorithm and the propagation delays in enforcing scheduling decisions.

3.2 Intra-Coflow Scheduling

To schedule flows belonging to the same coflow, we have designed a preemptive scheduling policy to help flows share the limited link bandwidth efficiently. Our schedul-

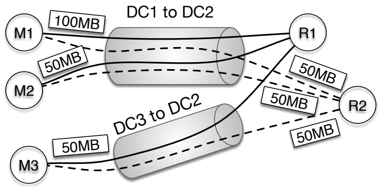


Figure 3: Network flows across datacenters in the shuffle phase of a simple job.

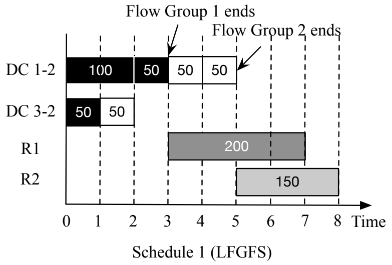


Figure 4: Job timeline with LFGF scheduling.

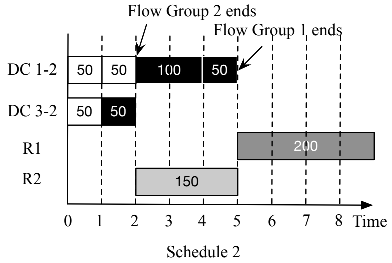


Figure 5: Job timeline with naive scheduling.

ing policy is called *Largest Flow Group First (LFGF)*, whose goal is to minimize *job* completion times. A *Flow Group* is defined as a group of all the flows that are destined to the same reduce task. The size of a flow group is the total size of all the flows within, representing the total amount of data received in the shuffle phase by the corresponding reduce task. As suggested by its name, LFGF preemptively prioritizes the flow group of the largest size.

The rationale of LFGF is to coordinate the scheduling order of flow groups so that the task requiring more computation can start earlier, by receiving their flows earlier. Here we assume that the task execution time is proportional to the total amount of data it received for processing. It is an intuitive assumption given no prior knowledge about the job.

As an example, we consider a simple Spark job that consists of two reduce tasks launched in datacenter 2, both requiring to fetch data from two mappers in datacenter 1 and one mapper in datacenter 3, as shown in Fig. 3. Corresponding to the two reducers *R1* and *R2*, two flow groups are sharing both inter-datacenter links, with the size of 200 MB and 150 MB, respectively. For simplicity, we assume the two links have the same bandwidth, and the calculation time per unit of data is the same as the network transfer time.

With LFGF, Flow Group 1, corresponding to *R1*, has a larger size and thus will be scheduled first. As is illustrated in Fig. 4, the two flows (*M1-R1*, *M2-R1*) in Flow Group 1 are scheduled first through the link between datacenter 1 and 2. The same applies to another flow (*M3-R1*) of Flow Group 1 on the link between datacenter 3 and 2. When Flow Group 1 completes at time 3, *i.e.*, all its flows complete, *R1* starts processing the 200 MB data received, and finishes within 4 time units. The other reduce task *R2* starts at time 5, processes the 150 MB data with 3 units of time, and completes at time 8, which becomes the job completion time.

If the scheduling order is reversed as shown in Fig. 5, Flow Group 2 will complete first, and thus *R2* finishes at time 5. Although *R1* starts at the same time as *R2* in Fig. 4, its execution time is longer due to its larger flow group size, which results in a longer job completion time. This example intuitively justifies the essence of LFGF — for a task that takes longer to finish, it is better to start it earlier by scheduling its flow group earlier.

3.3 Multi-Path Routing

Beyond ordering the coflows, we design a simple and efficient multi-path routing algorithm to utilize available link bandwidth better and to balance network load. The idea is similar to water-filling — it identifies the bottleneck link, and shifts some traffic to the alternative path with the lightest network load in an iterative fashion.

The bottleneck link is identified based on the time it takes to finish all the passing flows. In the first iteration, we calculate all the link load and the time it takes to finish all the passing flows, given that all the flows go through their direct links. To be particular, for each link l, the link load is represented as $D_l = d_i$, where d_i represents the total amount of data of the fetch i whose direct path is link l. The completion time is thus calculated as $t_l = D_l/B_l$, where B_l represents the bandwidth of link l. We identify the most heavily loaded link l^*, which has the largest t_{l^*}, and choose one of its alternative paths which has the lightest load for traffic re-routing. In order to compute the percentage of traffic to be rerouted from l^*, represented by α, we solve the equation $D_{l^*}(1-\alpha)/B_{l^*} = (D_{l^*}\alpha + D_{l'})/B_{l'}$, where l' is the link with the heaviest load on the selected detour path.

4 Siphon: Design and Implementation

4.1 Overview

To realize any coflow scheduling strategies in wide-area data analytics, we need a system that can flexibly enforce the scheduling decisions. Traditional traffic engineering [11, 14] techniques can certainly be applied, but they are not yet available to common cloud users. As is concluded in Sec. 2, Siphon is designed and implemented as a host-based building block to achieve this goal.

Fig. 6 shows a high-level overview of Siphon's architecture. Processes, called *aggregator daemons*, are deployed on all (or a subset of) workers, interacting with the worker processes of the data parallel framework directly. Conceptually, all these aggregators will form

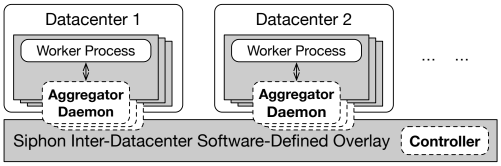


Figure 6: An architectural overview of *Siphon*.

an overlay network, which is built atop inter-datacenter WANs and supports the data parallel frameworks.

In order to ease the development and deployment of potential optimizations for inter-datacenter transfers, the Siphon overlay is managed with the software-defined networking principle. Specifically, aggregators operate as application-layer switches at the data plane, being responsible for efficiently aggregating, forwarding and scheduling traffic within the overlay. Network and flow statistics are also collected by the aggregators actively. Meanwhile, all routing and scheduling decisions are made by the central Siphon *controller*. With a flexible design to accommodate a wide variety of flow scheduling disciplines, the centralized controller can make fine-grained control decisions, based on coflow information provided by the resource scheduler of data parallel frameworks.

4.2 Data Plane

Siphon's data plane consists of a group of aggregator daemons, collectively forming an overlay that handles inter-datacenter transfers requested by the data parallel frameworks. Working as application-layer switches, the aggregators are designed with two objectives: it should be simple for data parallel frameworks to use, and supports high switching performance.

4.2.1 Software Message Switch

The main functionality of an aggregator is to work as a software switch, which takes care of fragmentizing, forwarding, aggregating and prioritizing the data flows generated by data parallel frameworks.

After receiving data from a worker in the data parallel framework, an aggregator will first divide the data into fragments such that they can be easily addressable and schedulable. These data fragments are called *messages*. Each data flow will be split into a sequence of messages to be forwarded within Siphon. A minimal header, with a flow identifier and a sequence number, will be attached to each message. Upon reaching the desired destination aggregator, they will be again reassembled and delivered to the final destination worker.

The aggregators can *forward* the messages to any peer aggregators as an intermediate nexthop or the final destination, depending on the forwarding decisions made by the controller. Inheriting the design in traditional OpenFlow switches, the aggregator looks up a forwarding table that stores all the forwarding rules in a hash table, to ensure high performance. Fortunately, wildcards in forwarding rule matching are also available, thanks to the hierarchical organizations of the flow identifiers. If neither the flow identifier nor the wildcard matches, the aggregator will consult the controller. A forwarding rule includes a nexthop to enforce routing, and a flow weight to enforce flow scheduling decisions.

Since messages forwarded to the same nexthop share the same link, we use a priority queue to buffer all pending outbound messages to support *scheduling* decisions. Priorities are allowed to be assigned to individual flows sharing a queue, when it is backlogged with a fully saturated outbound link. The control plane will be responsible for assigning priorities to each flows.

4.2.2 Performance-Centric Implementation

Since an aggregator is I/O-bounded, it is designed and implemented with performance in mind. It has been implemented in C++ from scratch with the event-driven asynchronous programming paradigm. Several optimizations are adopted to maximize its efficiency.

Event-driven design. events are raised and handled asynchronously, including all network I/O events. All the components are loosely coupled with one another, as each function in these components is only triggered when specific events it is designed to handle are raised. As examples of such an event-driven design, the switch will start forwarding messages in an input queue as soon as the queue raises a `PacketIn` event, and the output queue will be consumed as soon as a corresponding worker TCP connection raises a `DataSent` event, indicating that the outbound link is ready.

Coroutine-based pipeline design pattern. Because an aggregator may communicate with a number of peers at the same time, work conservation must be preserved. In particular, it should avoid head-of-line blocking, where one congested flow may take all resources and slow down other non-congested flows. An intuitive implementation based on input and output queues cannot achieve this goal. To solve this problem, our implementation takes advantage of a utility called "stackful coroutine," which can be considered as a procedure that can be paused and resumed freely, just like a thread whose context switch is controlled explicitly. In an aggregator, each received message is associated with a coroutine, and the total number of active coroutines is bounded for the same flow. This way, we can guarantee that non-congested flows can be served promptly, even coexisting with resource "hogs."

Minimized memory copying. Excessive memory copying is often an important design flaw that affects performance negatively. We used smart pointers and reference counting in our implementation to avoid memory

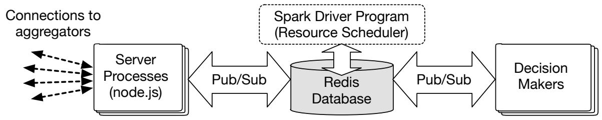

Connections to aggregators
Server Processes (node.js)
Pub/Sub
Spark Driver Program (Resource Scheduler)
Redis Database
Pub/Sub
Decision Makers

Figure 7: The architecture of the Siphon Controller.

copying as messages are forwarded. In the lifetime of a message through an aggregator, it is only copied between the kernel socket buffers for TCP connections and the aggregator's virtual address space. Within the aggregator, a message is always accessed using a smart pointer, and passed between different components by copying the pointer, rather than the data in the message itself.

4.3 Control Plane

The controller in Siphon is designed to make flexible control plane decisions, including flow scheduling and routing.

Although the controller is a logically centralized entity, our design objective is to make it highly scalable, so that it is easy to be deployed on a cluster of machines or VMs when needed. As shown in Fig. 7, the architectural design of the controller completely decouples the decision making processes from the server processes that directly respond to requests from Siphon aggregators, connecting them only with a Redis database server. Should the need arises, the decision making processes, server processes, and the Redis database can be easily distributed across multiple servers or VMs, without incurring additional configuration or management cost.

The Redis database server provides a reliable and high-performance key-value store and a publish/subscribe interface for inter-process communication. It is used to keep all the states within the Siphon datapath, including all the live statistics reported by the aggregators. The publish/subscribe interface allows server processes to communicate with decision-making processes via the Redis database.

The server processes, implemented in `node.js`, directly handle the connections from all Siphon aggregators. These server processes are responsible for parsing all the reports or requests sent from the aggregators, storing the parsed information into the Redis database, and responding to requests with control decisions made by the decision-making processes. It is flexible how the decision-making processes are implemented, depending on requirements of the scheduling algorithm.

In inter-coflow scheduling, the controller requires the full knowledge of a coflow before it starts. This is achieved by integrating the resource scheduler of the data parallel framework to the controller's Pub/Sub interface. Particularly in Spark, the task scheduler running in the driver program have such knowledge as soon as the reduce tasks are scheduled and placed on workers. We have modified the driver program, such that whenever there are new tasks being scheduled, the generated traffic information will be published to the controller. The incremental Monte Carlo simulations will then be triggered on the corresponding parallel decision makers.

5 Performance Evaluation

In this section, we present our results from a comprehensive set of experimental evaluations with Siphon, organized into three parts. First, we provide a coarse-grained comparison to show the application-level performance improvements by using Siphon. A comprehensive set of machine learning workloads is used to evaluate our framework compared with the baseline Spark. Then, we try to answer the question how Siphon expedite a single coflow by putting a simple shuffle under the microscope. Finally, we evaluate our inter-coflow scheduling algorithm, by using the state-of-the-art heuristic as a baseline.

5.1 Macro-Benchmark Tests

Experimental Setup. In this experiment, we run 6 different machine learning workloads on a 160-core cluster, which spans across 5 geographical regions. Performance metrics such as application runtime, stage completion time and shuffle read time are to be evaluated. The shuffle read time is defined as the completion time of the slowest data fetch in a shuffle. It reflects the time needed for the last task to start computing, and it determines the stage completion time to some extent.

The Spark-Siphon cluster. We set up a 160-core, 520 GB-memory Spark cluster. Specifically, 40 `n1-highmem-2` instances are evenly disseminated in 5 Google Cloud datacenter (N. Carolina, Oregon, Belgium, Taiwan, and Tokyo). Each instance provides 2 vCPUs, 13 GB of memory, and a 20 GB SSD of disk storage. Except for one instance in the N. Carolina region works as both Spark master and driver, all instances serve as Spark standalone executors. All instances in use are running Apache Spark 2.1.0.

The Siphon aggregators run on 10 of the executors, 2 in each datacenter. An aggregator is responsible for handling Pub/Sub requests from 4 executors in the same datacenter. The Siphon controller runs on the same instance as the Spark master, in order to minimize the communication overhead between them.

Note that we do not launch extra resources for Siphon aggregators to make the comparison fair. Even though they occupy some computation resource and system I/Os with their co-located Spark executors, the consumption is minimal.

Workload specifications. 6 machine learning workloads, with multiple jobs and multiple stages, are used

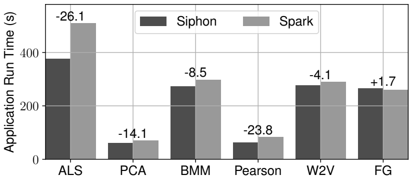
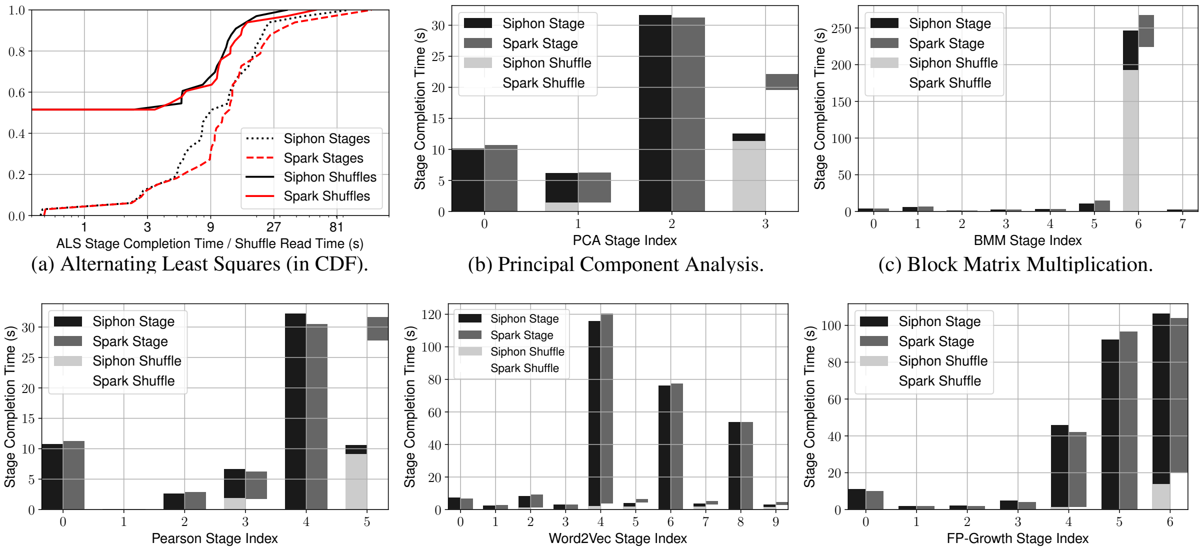


Figure 8: Average application run time.

Workload	# Shuffles	Total Bandwidth Usage(GB)	Extra Bandwidth Usage(MB)	Siphon Shuffle Read Time (s)	Spark Shuffle Read Time (s)	Runtime Reduction (%)	Cost Difference (¢)
ALS	18	40.47	2186.3	46.8	90.5	**48.3**	**-26.56**
PCA	2	0.51	37.6	3.3	13.7	**76.1**	**-6.80**
BMM	1	42.3	2911.1	48.9	97.8	**50.0**	**-29.26**
Pearson	2	0.57	23.8	3.6	13.1	**72.6**	**-6.23**
W2V	5	0.45	10.2	5.8	9.6	**39.9**	**-2.49**
FG	2	0.57	20.5	1.77	1.87	**5.4**	**-0.05**

Table 2: Summary of shuffles in different workloads (present the run with median application run time).



(a) Alternating Least Squares (in CDF).
(b) Principal Component Analysis.
(c) Block Matrix Multiplication.
(d) Pearson's Correlation.
(e) Word2Vec distributed presentation of words.
(f) FP-growth frequent item sets.

Figure 9: Shuffle completion time and stage completion time comparison (present the run with media application run time).

for evaluation.

- *ALS:* Alternating Least Squares.
- *PCA:* Principle Component Analysis.
- *BMM:* Block Matrix Multiplication.
- *Pearson:* Pearson's correlation.
- *W2V:* Word2Vec distributed presentation of words.
- *FG:* FP-Growth frequent item sets.

These workloads are the representative ones from Spark-Perf Benchmark[1], the official Spark performance test suite created by Databricks[2]. The workloads that are not evaluated in this paper share the same characteristics with one or more selected ones, in terms of the network traffic pattern and computation intensiveness. We set the scale factor to 2.0, which is designed for a 160-core, 600 GB-memory cluster.

Methodologies. With different workloads, we compare the performance of job executions, with or without Siphon integrated as its cross-datacenter data transfer service.

Note that, without Siphon, Spark works in the same way as the out-of-box, vanilla Spark, except one slight change on the `TaskScheduler`. Our modification eliminates the randomness in the Spark task placement decisions. In other words, each task in a given workload will be placed on a *fixed* executor across different runs. This way, we can guarantee that the impact of task placement on the performance has been eliminated.

Performance. We run each workload on the same input dataset for 5 times. The average application run time comparisons across 5 runs are shown in Fig. 8. Later we focus on job execution details, taking the run with median application run time for example. Table 2 summarizes the total shuffle size and shuffle read time of each workload. Further, Fig. 9 breaks down the time for network transfers and computation in each stage, providing more insight.

Among the 6 workloads, BMM, the most network-intensive workload, benefits most from Siphon. It enjoys a 23.6% reduction in average application run time. The reason is that it has one huge shuffle — sending more than 40 GB of data in one shot — and Siphon can help significantly. The observation can be proved by Fig. 9(c), which shows that Siphon manages to reduce around 50 seconds of shuffle read time.

Another network-intensive workload is ALS, an iterative workload. The average run time has been reduced by 13.2%. The reason can be easily seen with the information provided in Table 2. During a single run of the application, 40.47 GB of data is shuffled through the network, in 18 stages. Siphon collectively reduces the shuffle time by more than 30 seconds. Fig. 9(a) shows the CDFs of shuffle completion times and stage comple-

[1] `https://github.com/databricks/spark-perf`
[2] `https://databricks.com/`.

tion times, using Spark and Siphon respectively (note the *x*-axis is in log scale). As we observe, the long tail of the stage completion time distribution is reduced because Siphon has significantly improved the performance of all shuffle phases.

The rest of the workloads generate much less shuffled traffic, but their shuffle read time have also been reduced (5.4%∼76.1%).

PCA and Pearson are two workloads that have network-intensive stages. Their shuffle read time constitutes a significant share in some of the stages, but they also have computation intensive stages that dominate the application run time. For these workloads, Siphon greatly impacts the job-level performance, by minimizing the time used for shuffle (Table 2).

W2V and FG are two representative workloads whose computation time dominates the application execution. With these workloads, Siphon can hardly make a difference in terms of application run time, which is mostly decided by the computation stragglers. An extreme example is shown in Fig. 9(e). Even though the shuffle read time has been reduced by 4 seconds (Table 2), the computation stragglers in Stage 4 and Stage 6 will still slow down the application by 0.7% (Fig. 8). Siphon is not designed to accelerate these computation-intensive data analytic applications.

Cost Analysis. As the acceleration of Spark shuffle reads in Siphon is partially due to the relay of traffic through intermediate datacenters, it is concerned how it affects the overall cost for running the data analytics jobs. On the one hand, the relay of traffic increases the total WAN bandwidth usage, which is charged by public cloud providers. On the other hand, the acceleration of jobs reduces the cost for computation resources.

We present the total cost of running the machine learning jobs in Table 2, based on Google Cloud pricing[3]. Each instance used in our experiment costs \$1.184 per hour, and our cluster costs ¢ 0.6578 per second. As a comparison, the inter-datacenter bandwidth only costs 1 cent per GB.

As a result, Siphon actually reduced the total cost of running all workloads (Table 2). On the one hand, a small portion of inter-datacenter traffic has been relayed. On the other hand, the idle time of computing resources has been reduced significantly, which exceeds the extra bandwidth cost.

5.2 Single Coflow Tests

Experimental Setup. In the previous experiment, Siphon works well in terms of speeding up the coflows in complex machine learning workloads. However, one question remains unanswered: how does each component of Siphon contribute to the overall reduction on the coflow completion time? In this experiment, we use a smaller cluster to answer this question by examining a single coflow more closely.

The cross-datacenter Spark cluster consists of 19 workers and 1 master, spanning 5 datacenters. The Spark master and driver is on a dedicated node in Oregon. The geographical location of worker nodes is shown in Fig. 13, in which the number of executors in different datacenters is shown in the black squares. The same instance type (`n1-highmem-2`) is used.

Most software configurations are the same as the settings used in Sec. 5.1, including the Spark patch. In other words, the cluster still offers a fixed task placement for a given workload.

In order to study the system performance that generates a single coflow, we decided to use the `Sort` application from the HiBench benchmark suite [13]. `Sort` has only two stages, one map stage of sorting input data locally and a reduce stage of sorting after a full shuffle. The only coflow will be triggered at the start of the reduce stage, which is easier to analyze. We prepare the benchmark by generating 2.73 GB of raw input in HDFS. Every datacenter in the experiment stores an arbitrary fraction of the input data without replication, but the distribution of data sizes is skewed.

We compare the shuffle-level performance achieved by the following 4 schemes, with the hope of providing a comprehensive analysis of the contribution of each component of Siphon:

- *Spark:* The vanilla Spark framework, with fixed task placement decisions, as the baseline for comparison.
- *Naive:* Spark using Siphon as its data transfer service, without any flow scheduling or routing decision makers. In this case, messages are scheduled in a round-robin manner, and the inter-datacenter flows are sent directly through the link between the source to the destination aggregators.
- *Multi-path:* The *Naive* scheme with the multi-path routing decision maker enabled in the controller.
- *Siphon:* The complete Siphon evaluated in Sec. 5.1. Both LFGF intra-coflow scheduling and multi-path routing decision makers are enabled.

Job and stage level performance. Fig. 10 illustrates the performance of sort jobs achieved by the 4 schemes aforementioned across 5 runs, with respect to their job completion times, as well as their stage completion times for both map and reduce stages. As we expected, all 3 schemes using Siphon have improved job performance by accelerating the reduce stage, as compared to *Spark*. With *Naive*, the performance improvement is due to a higher throughput achieved by pre-established parallel TCP connections between Siphon aggregators.The im-

[3] `https://cloud.google.com/products/calculator/`

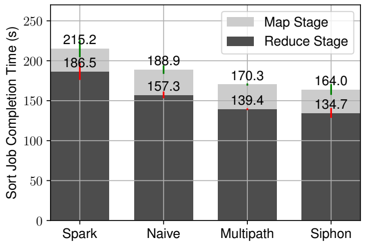


Figure 10: Average job completion time across 5 runs.

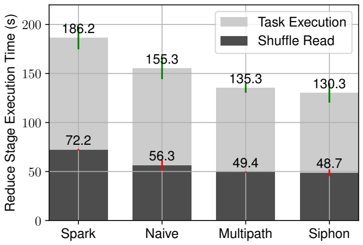


Figure 11: Breakdowns of the reduce stage execution across 5 runs.

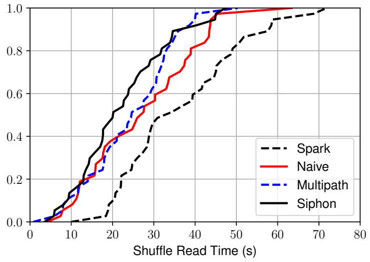


Figure 12: CDF of shuffle read time (present the run with median job completion time).

provement of *Multi-path* over *Naive* is attributed to a further reduction of reduce stage completion times — with multi-path routing, the network load can be better balanced across links to achieve a higher throughput and faster network transfer times. Finally, it is not surprising that *Siphon*, benefiting the advantages of both intra-coflow scheduling and *Multi-path* routing, achieves the best job performance.

To obtain fine-grained insights on the performance improvement, we break down the reduce completion time further into two parts: the shuffle read time (*i.e.,* coflow completion time) and the task execution time. As is shown in Fig. 11, the improvement of *Naive* over *Spark* is mainly attributed to a reduction of the shuffle read time.*Multi-path* achieves a substantial improvement of shuffle read time over *Naive*, since the network transfer completes faster by mitigating the bottleneck through multi-path routing. *Siphon* achieves a similar shuffle read time with *Multi-path*, with a slight reduction in the task execution time. This implies that multi-path routing is the main contributing factor for performance improvement, while intra-coflow scheduling helps marginally on the straggler mitigation as expected.

Shuffle: Spark v.s. Naive. To allow a more in-depth analysis of the performance improvement achieved by the baseline Siphon (*Naive*), we present the CDFs of shuffle read times achieved by *Spark* and *Naive*, respectively, in Fig. 12. Compared with the CDF of *Spark* that exhibits a long tail, all the shuffle read times are reduced by ∼10 s with *Naive*, thanks to the improved throughput achieved by persistent, parallel TCP connections between aggregators.

Shuffle: intra-coflow scheduling and multi-path routing. We further study the effectiveness of the decision makers, with *Multi-path* and *Siphon*'s CDFs presented in Fig. 12.

With multi-path routing enabled, both *Multi-path* and *Siphon* achieve shorter completion times (∼50 s) for their slowest flows respectively, compared to *Naive* (>60 s) with direct routing. Such an improvement is contributed by the improved throughput with a better balanced load across multiple paths. It is also worth noting that the percentage of short completion times achieved with *Multi-path* is smaller than *Naive* — 22% of shuffle reads complete within 18 s with *Multi-path*, while 35% complete with *Naive*. The reason is that by rerouting flows from bottleneck links to lightly loaded ones via their alternative paths, the network load, as well as shuffle read times, will be better balanced.

It is also clearly shown that with LFGF scheduling, the completion time of the slowest shuffle read is almost the same with that achieved by *Multi-path*. This meets our expectation, since the slowest flow will always finish at the same time in spite of the scheduling order, given a fixed amount of network capacity.

We further illustrate the inter-datacenter traffic during the sort job run time in Fig. 13, to intuitively show the advantage of multi-path routing. The sizes of the traffic between each pair of datacenters are shown around the bidirectional arrow line, the thickness of which is proportional to the amount of available bandwidth shown in Table 1.

The narrow link from Taiwan to S. Carolina becomes the bottleneck, which needs to transfer the largest amount of data. With our multi-path routing algorithm, part of the traffic will be rerouted through Oregon. We can observe that the original traffic load along this path is not heavy (only 149 MB from Taiwan to Oregon and 170 MB from Oregon to S. Carolina), and both alternate links have more available bandwidth. This demonstrates that our routing algorithm works effectively in selecting optimal paths to balance loads and alleviate bottlenecks.

5.3 Inter-Coflow Scheduling

In this section, we evaluate the effectiveness of Monte Carlo simulation-based inter-coflow scheduling algorithm, by comparing the average and the 90th-percentile Coflow Completion Time (CCT) with existing heuristics.

Testbed. To make the comparison fair, we set up a testbed on a private cloud, with 3 datacenters located in **V**ictoria, **T**oronto, and **M**ontreal, respectively. We have conducted a long-term bandwidth measurement among them, with more than 1000 samples collected for each link. Their distributions are depicted in Fig. 14, which are further used in the online Monte Carlo simulation.

Benchmark. We use the Facebook benchmark [8] workload, which is a 1-hour coflow trace from 150 work-

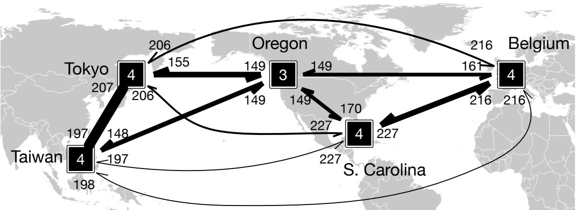


Figure 13: The summary of inter-datacenter traffic in the shuffle phase of the sort application.

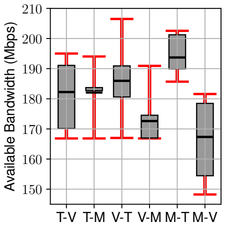


Figure 14: Bandwidth distribution among datacenters.

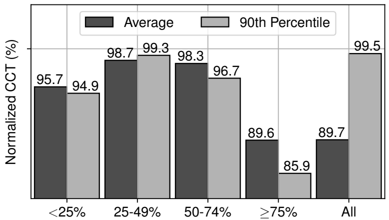


Figure 15: Average and 90th percentile CCT comparison.

ers. We assume workers are evenly distributed in the 3 datacenters, and generate aggregated flows on inter-datacenter links. To avoid overflow, the flow sizes are scaled down, with the average load on inter-datacenter links reduced by 30%.

Methodology. A coflow generator, together with a Siphon aggregator, is deployed in each datacenter. All generated traffic goes through Siphon, which can enforce proper inter-coflow scheduling decisions on inter-datacenter links. As a baseline, we experiment with the Minimum Remaining Time First (MRTF) policy, which is the state-of-the-art heuristic with full coflow knowledge [27]. The metrics CCTs are then normalized to the performance of the baseline algorithm.

Performance. Fig. 15 shows that Monte Carlo simulation-based inter-coflow scheduling outperforms MRTF in terms of both average and tail CCTs. Considering all coflows, the average CCT is reduced by ∼10%. Since the coflow size in the workload follows a long-tail distribution, we further categorize coflows in 4 bins, based on the total coflow size. Apparently, the performance gain mostly stems from expediting the largest bin – elephant coflows that can easily overlap with each other. Beyond MRTF, Monte Carlo simulations can carefully study all possible near-term coflow ordering with respect to the unpredictable flow completion times, and enforce a decision that is statistically optimal.

6 Related Work

Wide-Area Data Analytics. For data analytics spanning across datacenters, wide area network links easily become the performance bottleneck. To reduce the usage of inter-datacenter bandwidth, existing works either tweak applications to generate different workloads [12, 16, 23, 24], or assign input datasets and tasks to datacenters optimally [19, 22]. However, all these efforts focus on adding wide-area network awareness to the computation framework, without tackling the lower-level inter-datacenter data transfers directly. Orthogonal and complementary to these efforts, Siphon is designed for the inter-datacenter network optimization — it delivers the inter-datacenter traffic with better efficiency, regardless of the upper-layer decisions on task placement or execution plan.

Software-Defined Networking (SDN). The concept of SDN has been proposed to facilitate the innovation in network control plane [2, 18].In the inter-datacenter wide-area network, SDN has been recently adopted to provide centralized control with elegantly designed traffic engineering strategies [11, 14, 15]. Different from these efforts, our work realizes the SDN principle in the application layer, without requiring hardware support. Moreover, our work focuses on improving performance for data analytics jobs with more complex communication patterns, controlling flows at a finer granularity.

Network Optimization for Data Analytics. Accounting for the job-level semantics, coflow scheduling algorithms (*e.g.*, [6, 8, 9]) are proposed to minimize the average coflow completion time within a datacenter network, which is assumed to be free of congestion. Without such assumptions, joint coflow scheduling and routing strategies [17, 28] are proposed in the datacenter network, where both the core and the edge are congested. Different from these models, the network in the wide area has congested core and congestion-free edge, since the inter-datacenter links have much lower bandwidth than the access links of each datacenter. Apart from the different network model, our coflow scheduling handles the uncertainty of the fluctuating bandwidth in the wide area, while the existing efforts assume that the bandwidth capacities remain unchanged.

7 Concluding Remarks

To address the performance degradation of data analytics deployed across geographically distributed datacenters, we have designed and implemented Siphon — a building block that can be seamlessly integrated with existing data parallel frameworks — to expedite coflow transfers. Following the principles of software-defined networking, a controller implements and enforces several novel coflow scheduling strategies.

To evaluate the effectiveness of Siphon in expediting coflows as well as analytics jobs, we have conducted extensive experiments on real testbeds, with Siphon deployed across geo-distributed datacenters. The results have demonstrated that Siphon can effectively reduce the completion time of a single coflow by up to 76% and improve the average coflow completion time.

References

[1] Apache Hadoop Official Website. `http://hadoop.apache.org/`.

[2] Open Network Foundation Official Website. `https://www.opennetworking.org/`.

[3] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data Center TCP (DCTCP). In *Proc. ACM SIGCOMM* (2010).

[4] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pFabric: Minimal Near-Optimal Datacenter Transport. In *Proc. ACM SIGCOMM* (2013).

[5] Beck, J. C., and Wilson, N. Proactive Algorithms for Job Shop Scheduling with Probabilistic Durations. *Journal of Artificial Intelligence Research 28* (2007), 183–232.

[6] Chowdhury, M., and Stoica, I. Efficient Coflow Scheduling Without Prior Knowledge. In *Proc. ACM SIGCOMM* (2015).

[7] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M., and Stoica, I. Managing Data Transfers in Computer Clusters with Orchestra. In *Proc. ACM SIGCOMM* (2011).

[8] Chowdhury, M., Zhong, Y., and Stoica, I. Efficient Coflow Scheduling with Varys. In *Proc. ACM SIGCOMM* (2014).

[9] Dogar, F. R., Karagiannis, T., Ballani, H., and Rowstron, A. Decentralized Task-Aware Scheduling for Data Center Networks. In *Proc. ACM SIGCOMM* (2014).

[10] Hong, C. Y., Caesar, M., and Godfrey, P. B. Finishing Flows Quickly with Preemptive Scheduling. In *Proc. of ACM SIGCOMM* (2012).

[11] Hong, C.-Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., and Wattenhofer, R. Achieving High Utilization with Software-Driven WAN. In *Proc. of ACM SIGCOMM* (2013).

[12] Hsieh, K., Harlap, A., Vijaykumar, N., Konomis, D., Ganger, G. R., Gibbons, P. B., and Mutlu, O. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[13] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *Proc. International Conference on Data Engineering Workshops (ICDEW)* (2010).

[14] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *Proc. of ACM SIGCOMM* (2013).

[15] Jin, X., Li, Y., Wei, D., Li, S., Gao, J., Xu, L., Li, G., Xu, W., and Rexford, J. Optimizing Bulk Transfers with Software-Defined Optical WAN. In *Proc. of ACM SIGCOMM* (2016).

[16] Kloudas, K., Mamede, M., Preguiça, N., and Rodrigues, R. Pixida: Optimizing Data Parallel Jobs in Wide-Area Data Analytics. *VLDB Endowment 9*, 2 (2015), 72–83.

[17] Li, Y., Jiang, S. H.-C., Tan, H., Zhang, C., Chen, G., Zhou, J., and Lau, F. Efficient Online Coflow Routing and Scheduling. In *Proc. ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)* (2016).

[18] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR 38*, 2 (2008), 69–74.

[19] Pu, Q., Ananthanarayanan, G., Bodik, P., Kandula, S., Akella, A., Bahl, P., and Stoica, I. Low Latency Geo-Distributed Data Analytics. In *Proc. ACM SIGCOMM* (2015).

[20] Rasley, J., Stephens, B., Dixon, C., Rozner, E., Felter, W., Agarwal, K., Carter, J., and Fonseca, R. Planck: Millisecond-Scale Monitoring and Control for Commodity Networks. In *Proc. of ACM SIGCOMM* (2014).

[21] Susanto, H., Jin, H., and Chen, K. Stream: Decentralized Opportunistic Inter-Coflow Scheduling for Datacenter Networks. In *Proc. IEEE International Conference on Network Protocols (ICNP)* (2016).

[22] Viswanathan, R., Ananthanarayanan, G., and Akella, A. Clarinet: Wan-Aware Optimization for Analytics Queries. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[23] Vulimiri, A., Curino, C., Godfrey, P., Jungblut, T., Padhye, J., and Varghese, G. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[24] Vulimiri, A., Curino, C., Godfrey, P., Karanasos, K., and Varghese, G. WANalytics: Analytics for a Geo-Distributed Data-Intensive World. In *Proc. Conference on Innovative Data Systems Research (CIDR)* (2015).

[25] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012).

[26] Zats, D., Das, T., Mohan, P., Borthakur, D., and Katz, R. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proc. ACM SIGCOMM* (2012).

[27] Zhang, H., Chen, L., Yi, B., Chen, K., Chowdhury, M., and Geng, Y. CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark. In *Proc. ACM SIGCOMM* (2016).

[28] Zhao, Y., Chen, K., Bai, W., Yu, M., Tian, C., Geng, Y., Zhang, Y., Li, D., and Wang, S. Rapier: Integrating Routing and Scheduling for Coflow-Aware Data Center Networks. In *Proc. IEEE INFOCOM* (2015).

PerfIso: Performance Isolation for Commercial Latency-Sensitive Services

Călin Iorgulescu* *EPFL*
Reza Azimi* *Brown University*
Youngjin Kwon* *U. Texas at Austin*
Sameh Elnikety *Microsoft Research*
Manoj Syamala *Microsoft Research*
Vivek Narasayya *Microsoft Research*
Herodotos Herodotou* *Cyprus University of Technology*
Paulo Tomita *Microsoft Bing*
Alex Chen *Microsoft Bing*
Jack Zhang *Microsoft Bing*
Junhua Wang *Microsoft Bing*

Abstract

Large commercial latency-sensitive services, such as web search, run on dedicated clusters provisioned for peak load to ensure responsiveness and tolerate data center outages. As a result, the average load is far lower than the peak load used for provisioning, leading to resource under-utilization. The idle resources can be used to run batch jobs, completing useful work and reducing overall data center provisioning costs. However, this is challenging in practice due to the complexity and stringent tail-latency requirements of latency-sensitive services. Left unmanaged, the competition for machine resources can lead to severe response-time degradation and unmet service-level objectives (SLOs).

This work describes *PerfIso*, a performance isolation framework which has been used for nearly three years in Microsoft Bing, a major search engine, to colocate batch jobs with production latency-sensitive services on over 90,000 servers. We discuss the design and implementation of *PerfIso*, and conduct an experimental evaluation in a production environment. We show that colocating CPU-intensive jobs with latency-sensitive services increases average CPU utilization from 21% to 66% for off-peak load without impacting tail latency.

1 Introduction

New server acquisition contributes to over half of the total cost of ownership (TCO) of modern data centers [8]. However, server utilization is low in data centers hosting large latency-sensitive services for two main reasons: First, latency-sensitive services are typically provisioned for the peak load, which occurs only for a fraction of the total running time [18]. Second, business-continuity plans dictate tolerating multiple major data center outages, such as tolerating the failure of two data centers out of three data centers within a continent while remaining capable of processing peak load. The high degree of over-provisioning is imperative: a livesite incident causing brief downtime results in lost revenue and frustrated users, while an extended downtime comes with negative headline news and irreparable business damage. Even slightly higher response times decrease user satisfaction and impact revenues [29, 10, 17].

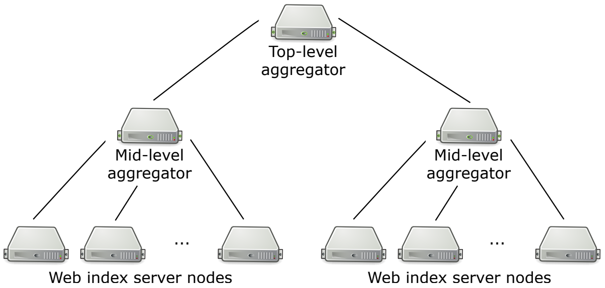


Figure 1: Architecture of index serving system of Web search engine with two aggregation levels (MLA and TLA). The user query is processed on index servers, which send responses to MLAs, which send aggregated responses to TLA.

Over-provisioning means that resource utilization is low, offering the opportunity to colocate batch jobs alongside latency-sensitive services [32, 18]. Colocation must be managed carefully lest it degrades performance due to competition on machine resources. Our main goal is to ensure that the end-to-end service-level objectives (SLOs) are met while increasing the work done by batch jobs. The main technical challenges arise from maintaining short tail latency (e.g., the 99^{th} latency percentile also called P99 latency) for the latency-sensitive services coupled with the complexity of commercial software and large deployments.

Oftentimes the service-level-objectives are not known explicitly for each individual component. For example, large commercial search engines contain tens of plat-

[*] Work done while authors were at Microsoft Research.

forms: to serve the web index, to build and update the web index, to manage user data and transaction history, to serve the most relevant advertisements, and to bill advertisers among many others. Modeling these components or assuming all their target latency values are known is not realistic.

Production environments are complex. A large data center comprising over 100,000 machines spans several generations of hardware. The generation gap can be up to 6 years, effectively constraining which hardware features can be used for performance isolation. Changes to the software stack running in a production environment are often infeasible. To be deployed on a large scale, the performance isolation framework must be robust, modular, and easy to debug. A good solution must provide the same performance guarantees seamlessly across all hardware and software configurations.

We describe our experiences in developing and deploying *PerfIso*, the performance isolation framework used in Microsoft's Bing clusters for over three years. We show how to colocate batch jobs with online services even when the tail response-latency requirements are within the order of milliseconds. We describe *CPU blind isolation* which dynamically restricts the cores that batch jobs use to protect the bursty interactive services even under high load. Depending on the load, batch jobs are given more or fewer resources to make progress.

Existing colocation approaches [20, 16, 34, 38] measure server-level performance metrics (e.g., query response times), and adjust resource allocation when the target is not met. This is not a good fit because if a query misses its target, it is already too late [17], and only end-to-end response time constraints are specified; per-layer service time limits are not.

We take a different approach: we ensure that there is always some slack in available resources such that abrupt changes in load do not impact response times. In contrast, traditional resource management polices focus on high resource-utilization while enforcing fair-sharing (most operating systems employ work-conserving scheduling algorithms). This works well for batch jobs, but does not account for factors such as the response-time latency of an interactive service. By using non-work-conserving resource management, we are able to adapt to changes in load and resource demands while treating the latency-sensitive service as a "black-box".

We focus on a concrete example: IndexServe — the Web index serving platform — because it is one of the largest in terms of machine count and has some of the strictest latency requirements. The web index is partitioned across hundreds or thousands of servers, and a user search query is processed in parallel on all servers. Responses are aggregated from the IndexServe machines on multiple levels (see Fig. 1). In such multi-layered systems, the slowest server dictates the response time [15].

To handle high load while meeting the strict latency requirements, many services are implemented as highly-optimized multi-threaded servers. The low query servicing-times make them highly bursty in nature: in several Bing services we find that, under high load, up to 15 threads become ready to run in just $5\mu s$. Due to the stringent tail-latency constraints, it is imperative to avoid scheduling delays, making the CPU the main bottleneck in our approach. We show that statically restricting CPU cores or CPU cycles does not fully solve the problem and fails to take advantage of idle cores during off-peak.

Our key goal is to ensure that interactive services perform equally well with batch jobs colocated as when they run alone. We show that CPU blind isolation successfully protects IndexServe while increasing average CPU utilization from 21% to 66% by colocating it with CPU-intensive jobs.

The main contributions of this work are as follows:

i. Identifying the key challenges of colocating batch jobs with large production latency-sensitive services and analyzing the effectiveness of operating system mechanisms to monitor and control resources.

ii. Introducing *CPU Blind Isolation* — a technique to mitigate harmful CPU-level interference between tenants.

iii. Designing and implementing the *PerfIso* performance isolation framework which allows batch jobs to be run alongside latency-sensitive services without any tail latency degradation.

iv. Evaluating *PerfIso* on a single machine to compare it to other alternatives, and on a 75-node production cluster, both running a real-world commercial online interactive service.

2 Background

We refer to the latency-sensitive user-facing services as *primary tenants*. All resources of a machine need to be available for them, since they generate the revenue that pays for the actual machines. The main goal of our system is to colocate batch jobs with a latency-sensitive user-facing service without impacting its response times. Thus, the primary always runs <u>unrestricted</u> and <u>unmodified</u>.

Batch jobs that run on these machines are *secondary tenants* and are treated in a best-effort manner — any resources they use are released to the system whenever the *primary* needs them. If the primary does not utilize all available resources, the secondary will be allowed to use some of them.

2.1 Applications and Services

Primary tenants comprise the latency-sensitive services which are governed by strict response-time SLOs. They are characterized by the following:

1. **a complex layered architecture** – hard to model or predict, since responses are computed in parallel and then aggregated.

2. **short tail latency** – any layer can severely impact query response times.

3. **highly bursty nature** – the service frequently spawns a large number of workers in a short period of time (order of microseconds).

Example primary tenant. We take the IndexServe component of **Bing** search as an example. IndexServe receives user queries and fetches potential matches from a search index. It can perform a variety of lookup and ranking operations. Its response times are within the order of milliseconds, and SLOs dictate that the 99^{th} percentile must stay within a 1-millisecond limit of its expected value (i.e., without colocation). Fig. 1 shows a simplified description of the layered architecture of IndexServe. Our measurements indicated that during a time frame of $5\mu s$, up to 15 threads became ready for execution.

Example secondary tenants. Non-latency-sensitive, big-data applications run alongside the primary. Popular big-data frameworks such as Hadoop [1], YARN [31], Apache Spark [35, 7], or Apache Flink [9] allow running a wide-range of compute-intensive (e.g., machine learning), and disk-bound (e.g., search index preparation and aggregation) jobs. Additionally, each server needs to run an *HDFS DataNode* process (for data replication), and a *YARN NodeManager* process (to handle individual task creation/destruction).

Both classes of tenants require access to several resources: CPU, disk, memory, network, etc. Accounting for potential resource bottlenecks is paramount in maintaining the performance of the primary. However, that alone is not sufficient, as our system needs to ensure that the secondary can make adequate progress, increasing the amount of work done when the primary is underutilized.

2.2 Driving Forces and Constraints

We focus on the performance requirements of the primary, without making any assumptions about its implementation. This enables wide-spread deployment, but makes it difficult to identify when the secondary interferes with the primary. Although we consider the CPU the main bottleneck, other resources also need to be monitored for contention.

Another key aspect is controlling resource access. Most operating systems already offer static mechanisms to restrict or prioritize access to a certain resource. While comprehensive, these are insufficient when dealing with bursty latency-sensitive workloads.

Given the complexity of production systems, it is hard to change the primary or the operating system (especially the kernel) due to the high costs of development, testing and deployment. Rather, our solution relies on features readily-available, and makes very few assumptions about the primary workload. In a nutshell, we treat the primary and the OS as a “black-box”.

3 System Design

3.1 CPU Blind Isolation

CPU is a prevalent bottleneck resource for most low-latency services and big-data frameworks [27]. Modern OSes implement effective means to statically manage CPU time across tenants [3, 4], such as throttling CPU cycles or restricting CPU cores. However, in Section 6.1.4 we show that they are ineffective because they cannot automatically adapt to the bursty workloads.

We propose a new technique called *CPU Blind Isolation*, which restricts which cores secondary tenants use based on core utilization information read from the OS. The key idea is to ensure that the primary *always has some headroom* (i.e., **buffer idle cores**), to absorb any primary worker-threads that wake up. The number of buffer cores is computed after offline-profiling of the primary using a sufficiently-heavy workload.

As the primary must always run unrestricted, we *restrict the secondary* to run only on a subset of cores. The secondary is allocated the cores remaining after subtracting the cores used by the primary and the number of buffer idle cores.

For example, consider a machine with 48 (physical) cores running a primary that needs 4 buffer cores to absorb bursts. If the primary uses 20 cores, the secondary would be restricted to 24 cores. If the primary goes up to 24 cores, the secondary is immediately restricted to 20.

Why not change the OS scheduler? We recognize that this solution can be implemented at the scheduler level. We argue that this is impractical and imposes significant overhead in large-scale deployment. Bugs introduced to the scheduler by seemingly-trivial changes are laborious to track down and can cause unexpected performance degradation. For example, the well-established Linux Completely-Fair-Scheduler has been found to have had bugs which caused threads to wait even when idle cores were available [21]. These bugs persisted in the codebase for several years. Our approach achieves performance isolation without interfering with the scheduler or the scheduling policy.

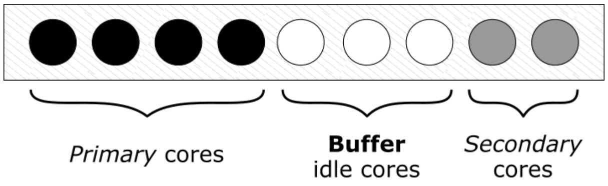


Figure 2: Conceptual representation of *CPU blind isolation*. The primary is unrestricted, and can run on any core, while the secondary is restricted to a subset of cores such that the primary always has a buffer of idle cores.

Non-work conserving scheduling In contrast to most OS schedulers, *blind isolation* is non-work conserving by nature. It deliberately chooses to leave several cores idle in order to properly measure and react to changes in the amount of work done by the primary. It is known that non-work conserving schedulers can improve performance in multi-processor scenarios [13]. We find that, similarly, blind isolation helps improve CPU utilization in the case of colocation.

3.1.1 Counting Idle Cores

An important requirement of our solution is a *low-latency*, *low-overhead* means of obtaining CPU utilization information. More specifically, we need to know how many cores are idle. We consider a core to be idle if the *idle thread* is running there.

The Windows scheduler keeps track of idle cores and provides this information through a system call. This system call returns a bit mask with the bits corresponding to the idle CPUs' ids set. We tested several other approaches relying on different metrics (e.g., recorded idle times, counting active threads), but found this solution to be best in terms of latency, overhead, and accuracy.

3.1.2 Allocating Cores to the Secondary

Once we know how many cores are idle we can detect whether the secondary needs to give up cores to the primary, or if the primary is not using all available cores. We assume that the secondary is CPU-intensive, and thus will fully occupy all cores allocated to it. If **I** is the number of idle cores in the system, **B** is the number of buffer cores, and **S** is the number of cores allocated to the secondary, then: if $\mathbf{I} < \mathbf{B}$, **S** is decreased, and if $\mathbf{I} > \mathbf{B}$, **S** is increased.

3.2 Managing Other Resources

Disk. We *choose which disks* are best suited for the primary and secondary. We find that it is necessary to separate the disks which are on the critical path of the primary from those used by the secondary. This is motivated by the nature of the tenants: the primary is highly-tuned towards read-only random accesses, so it is assigned to a striped set of solid-state disks (SSDs). In contrast, batch jobs often perform both reads and writes and mostly sequential in nature, so they are assigned to a striped set of hard-disks (HDDs).

Memory. Most low-latency services will manage their caches explicitly, loading data from disk as necessary depending on incoming queries. Furthermore, primary services are engineered to have a fixed working set and a stable memory footprint. We cannot compromise on this, and must guarantee the primary's ability to make full use of the memory. This is achieved by *limiting the memory footprint* of the secondary. When memory runs very low, secondary processes are killed.

Egress network packets. We *throttle* the outbound traffic of the secondary, marking it as low-priority and allowing the primary to maintain its throughput and response latency. This prevents the secondary from affecting the responsiveness of the primary.

4 Implementation and Deployment

We implemented *PerfIso* as a user-mode service based on the techniques and OS mechanisms described. Most of the static limits that *PerfIso* enforces are read from cluster-wide configuration files distributed through the Autopilot [14] environment. The resource limits can be altered independently at runtime by issuing a command to *PerfIso*. A client-application can also be used locally for debugging.

Although it is possible to obtain the unique process identifiers (PIDs) of the secondary tenants, Autopilot eases this task by keeping a list of running services and their respective information. Each secondary tenant process is placed in a unified *Job Object* configured dynamically by *PerfIso*.

4.1 Isolation Algorithm

The dynamic limits set by *PerfIso* need to be adjusted often. The state of the system needs to be read and the control knobs updated accordingly. Polling is important because the state of the primary can change quickly. Unfortunately, constantly updating certain settings can become harmful to the performance of all services. Thus, *polling and updating are separated* in *PerfIso*. We poll utilization data (e.g., CPU) continuously in a tight loop and we update the dynamic limits of the system on-demand based on the measured change in resource requirements.

Choosing the number of buffer cores. *CPU blind isolation* uses buffer cores to ensure that tail latency is protected while the system adjusts to changes in load. This requires that sufficient buffer cores are allocated to absorb bursty workloads. A one-off measurement of the primary under its provisioned peak load is needed to find how many threads can become ready for execution.

We evaluate different buffer core values in Section 6.1.3, and find that 8 cores are enough for IndexServe to maintain its 99th percentile SLO on our servers.

I/O throttling. The monitoring mechanisms provide only per-device I/O statistics, without discerning which processes originated the operations. In order to provide per-process throttling of I/O, we use *Deficit-Weighted-Round-Robin* [19]. Each process is assigned an I/O priority and one or more limits that need to be enforced (e.g., bandwidth, IOPS). Based on its priority, each process is assigned a weight – the higher the priority, the larger the weight. We then measure the number of completed I/O requests per second (or IOPS) per drive, and use a moving average.

We compute the portion of the requests a given process is responsible for based on its weight. Considering w_i^t the weight of process i at time t, and $curr^t$ the IOPS value measured at time t, then the demand of process i is:

$$D_i^t = \sum_{t'=t-\Delta}^{t} \frac{w_i^{t'} \times curr^{t'}}{\sum_{\forall j} w_j^{t'}}$$

We mark the lower limit of process i with lim_i, which represents the minimum amount of IOPS that process i is guaranteed. The deficit of this process with regard to the limit is:

$$\mathrm{Def}_i^t = \frac{curr^t - min(lim_i, D_i^t)}{min(lim_i, D_i^t)}$$

The I/O priorities of processes are adjusted based on the computed deficit values.

4.2 Deployment in Production Clusters

All machines run under a management framework such as Autopilot [14]. This provides machine wiping, imaging, backup, and monitoring functionality. Autopilot provides a stable service management interface to start, stop, and configure software. *PerfIso* is run as an additional service in Autopilot, making it easy to deploy, and to configure across various different environments.

PerfIso is designed to have a “kill-switch”, so that it can be quickly deactivated. This is useful when debugging production issues, and it allows quickly excluding *PerfIso* as a potential cause.

PerfIso is fully recoverable, since all parameters are stored in the cluster-wide configuration files. In the event of a crash, Autopilot will bring it up again, and *PerfIso* will resume its function by loading its state from disk.

PerfIso ensures that its settings do not affect those employed by the primary. For example, if the primary uses core affinitization for performance reasons, then *PerfIso* would not override these settings when attempting to accommodate the secondary.

5 Experimental Evaluation

5.1 Objectives

1. How is tail latency impacted by colocating batch jobs with the primary without *PerfIso*?

2. How effective is *PerfIso* in maintaining tail latency when a batch job is colocated with the primary?

3. How does CPU *blind isolation* compare to static isolation mechanisms provided by modern OSes?

5.2 Machine Configuration

We evaluate our solution on typical production hardware. Each server has two Intel Xeon E5-2673 v3 processors with 12 physical cores per die (a total of 48 cores with hyper-threading), 128 GB of RAM, and a 10 GbE Ethernet card. Storage is provided by 2 striped volumes: 4× 500 GB SSD drives, and 4× 2 TB HDD drives. The servers run Microsoft Windows Server 2016.

5.3 Experiment Setup

We use Bing IndexServe as the **primary** tenant in our experiments. IndexServe processes a search query to find a match using a large index partitioned across machines and replicated for performance.

IndexServe is setup with an index slice of 569 GB, and uses approximatively 110 GB of memory to cache recently accessed web index data. The index slice is stored on the striped SSD volume which IndexServe uses exclusively. IndexServe relies on the SSDs’ low I/O latency to maintain its tail latency requirements. The HDD volume is only used by IndexServe for logging, being shared with the secondary. The service is configured to return the most relevant matches.

Primary workload. We use a trace of 500k real-world queries from early 2017 to put load on the primary. A separate client machine is used to submit queries from the trace. We first replay a warm-up trace of 100k queries at a rate of 300 queries / second (QPS) so that IndexServe ramps up and reaches a steady-state. The warm-up is not reported as part of our measurements.

We vary the load by changing the query arrival rate, *i.e.*, we replay our trace at different query rates. The following represent a reasonable approximation of query arrival rates that an IndexServe machine might receive at the time that this paper was written:

- 2,000 QPS - approximating average load

- 4,000 QPS - approximating peak load

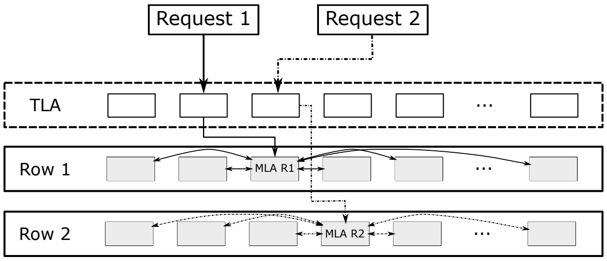


Figure 3: Request processing on the 75 machine IndexServe cluster described in Section 5.3. All gray machines run IndexServe and hold a slice of the index.

The client application replays the query trace in an open loop and sends queries according to a Poisson process distribution.

Secondary workload. We use a synthetic microbenchmark as the **secondary** to stress the CPU by actively utilizing as many CPU cycles as the system permits, pushing *blind isolation* to its limits. This **CPU bully** is a multi-threaded program with each worker thread computing the sum of several integer values. The number of worker threads is configurable and we vary it up to the total number of logical cores present on the system. The bully maximizes CPU utilization since there are very few memory or external storage accesses.

Single-machine experiments. We run IndexServe on a single machine configured as described above. We measure the impact of CPU contention on tail query response time, the most important metric being the 99th percentile.

Cluster experiments. We setup IndexServe across 75 machines, in the following manner: the index is split into 22 partitions (or *columns*), and each column is replicated by a factor of 2 (total of 2 *rows*). Each IndexServe server holds a partition of the index similarly to the single-box runs. The top-level aggregator (TLA) runs on 31 separate machines than the ones that hold the index. The mid-level aggregator (MLA) runs on IndexServe machines, and each request may get forwarded to a different MLA based on the TLA load-balancing. Fig. 3 shows an example of the system processing 2 incoming requests.

Each IndexServe machine also runs an HDFS client because many batch jobs that are used in production run on top of frameworks such as Hadoop and, thus, rely on HDFS for storage access. In addition to other experiment-specific *PerfIso* settings, we also set the following static disk bandwidth limits: data replication is limited to 20MB/s, and HDFS clients are limited to 60 MB/s. All I/O operations done by HDFS are unbuffered.

A client is setup on a separate machine and configured to submit queries to the TLA machines. We then run a trace of 200k queries at a total rate of 8,000 QPS. The TLAs will load-balance these queries across the 2 rows, resulting in an average workload of 4,000 QPS for each IndexServe machine.

Additionally, we use a **Disk bully** to ensure that I/O generated by HDFS does not cause any server to straggle. We setup DiskSPD [5] to create an I/O bound workload on the HDD strip of each machine. We perform a mixed read-write workload, with 33% reads and 67% writes, with sequential accesses and synchronous I/O operations.

6 Experimental Results

We first evaluate *PerfIso* on a single machine and measure the effectiveness of CPU blind isolation. We then move on to a 75-machine cluster and analyze CPU isolation mechanisms, measuring latency end-to-end and at each component level. The main metric used is the 99th percentile of query response latency.

6.1 Single-machine Experiments

Going further, we analyze the baseline (or standalone) behaviour of IndexServe and three colocation scenarios:

- **No isolation** – The primary and secondary are colocated without any isolation.
- **Blind isolation** – The secondary is dynamically restricted in terms of CPU cores using our technique.
- **Alternative isolation** – The secondary is successively restricted in terms of CPU cores, and CPU cycles using OS-specific mechanisms.

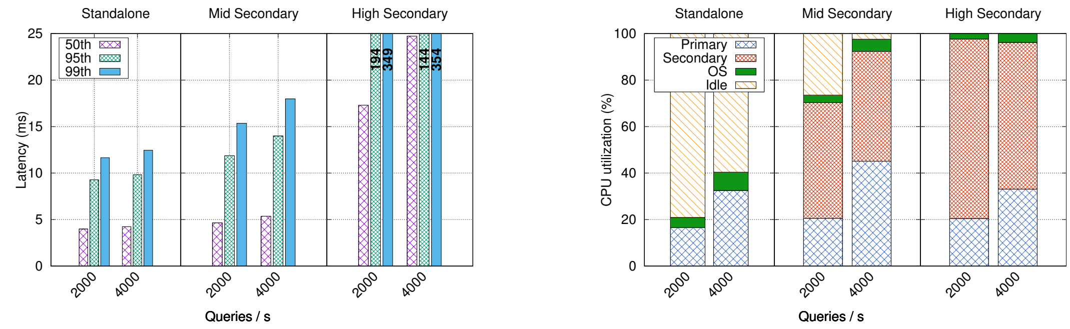
Our goal is to also maximize the amount of work done by the secondary, so we first configure each isolation technique with "relaxed" settings. We then successively restrict the secondary until either the SLO is met, or until the secondary no longer gets any work done.

6.1.1 Baseline

First, we measure how IndexServe performs when it runs standalone (i.e., no colocation). The 1st bar groups of Figs. 4a and 4b report the query latency and CPU utilization, respectively. The median query time is 4ms, and the 99th percentile is 12ms, both for 2,000 and 4,000 QPS. The average CPU utilization is low, with the CPU remaining idle for 80% and 60% of time, respectively.

6.1.2 No Isolation

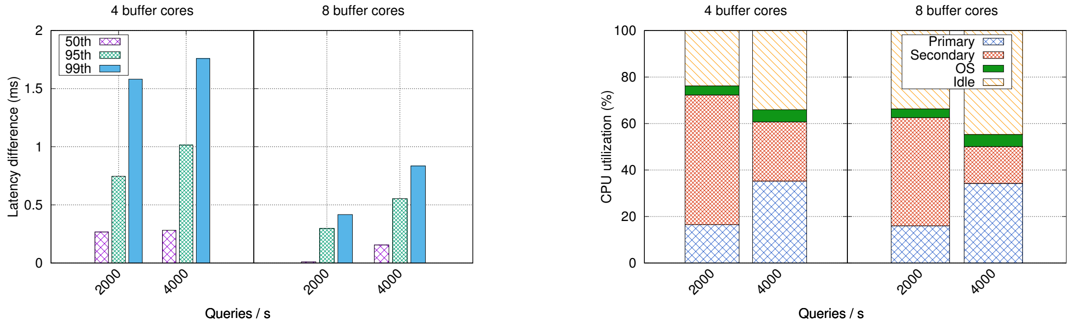
We colocate the primary and secondary, configuring the bully to use either *mid* (24 threads) or *high* (48 threads). The 2nd and 3rd columns of Figs. 4a and 4b report query latency and CPU utilization for the *mid* and



(a) Query response latency

(b) CPU utilization

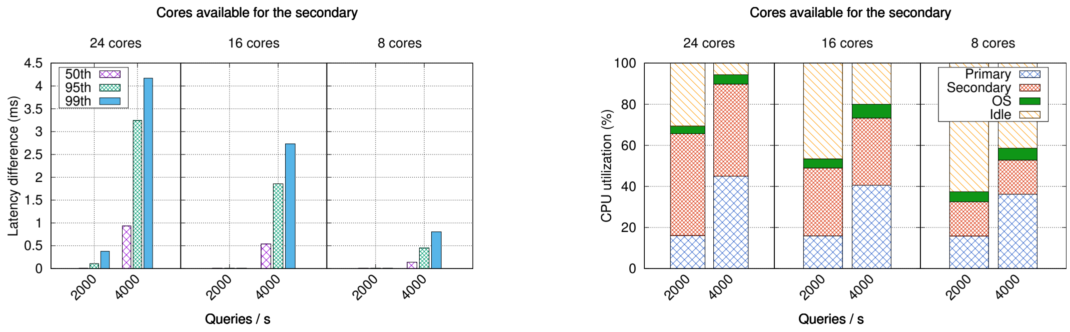
Figure 4: Single machine run of IndexServe standalone (no colocation) vs. colocated with an unrestricted secondary. A *mid* secondary increases the 99th percentile query latency by up to 42%, while a *high* increases same by up to 29×.



(a) Query latency degradation

(b) CPU utilization

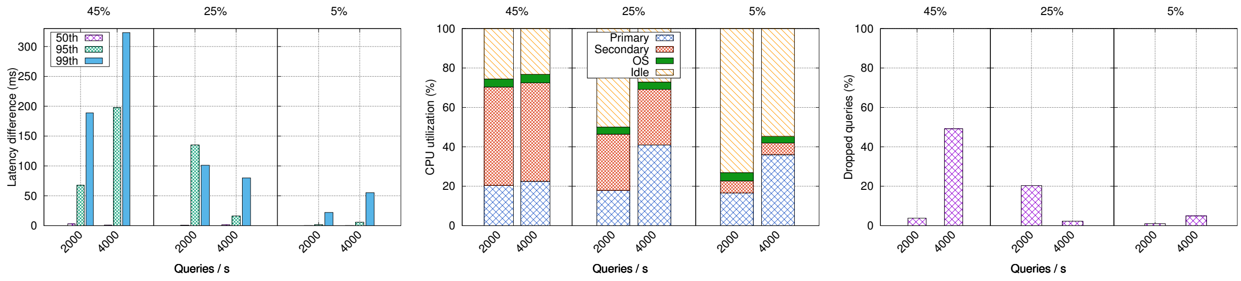
Figure 5: Single machine run of IndexServe colocated with a secondary restricted using blind isolation. Using a buffer of 8 CPU cores, the 99th percentile query latency is less than 1 ms off from the standalone case.



(a) Query latency degradation

(b) CPU utilization

Figure 6: Single machine run of IndexServe colocated with a secondary with CPU cores statically restricted.

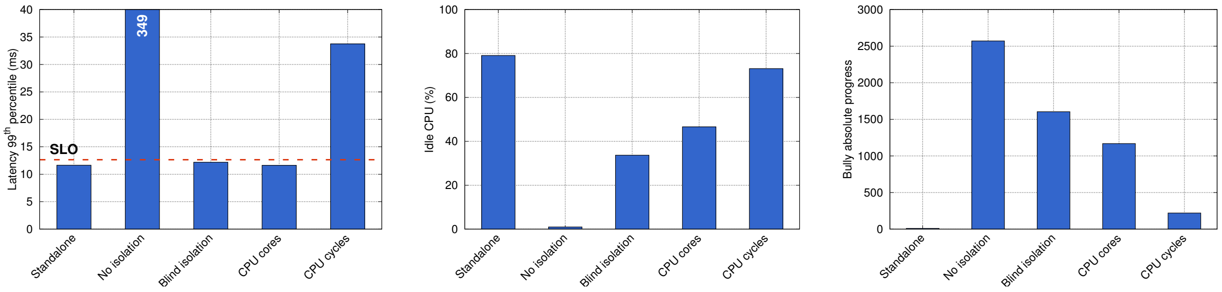


(a) Query latency degradation

(b) CPU utilization

(c) Dropped queries

Figure 7: Single machine run of IndexServe colocated with a secondary with CPU cycles statically restricted.



(a) Query latency

(b) Idle CPU

(c) Secondary progress

Figure 8: Comparison of isolation approaches on a single machine run of IndexServe at 2,000 QPS colocated with a secondary running in *high*-mode. CPU blind isolation uses 8 buffer cores, CPU cores allows the secondary to use 8 cores, and CPU cycles restricts the secondary to 5% of CPU time.

high configurations, respectively. The impact of colocation is substantial. The *mid* case reaches 15ms and 18ms in the 99th percentile for 2,000 and 4,000 QPS (higher than the baseline by 3-5ms). For *high*, these values reach 349ms and 354ms (a 29× degradation). Between 11% and 32% of queries timeout.

Fig. 4b shows the CPU utilization for the primary and secondary, and idle CPU. Interestingly, when colocated with the *mid* secondary, the CPU utilization of the primary increases up to 40% — IndexServe tries to compensate for the increase in pending queries by starting more workers. While this successfully prevents dropped queries, it ultimately aggravates CPU contention, and the latency SLO is not met. Another consequence is that the secondary gets less CPU time overall, since the primary will push it out from the only cores where it can run.

In the case of the *high* secondary, more than 32% of queries submitted at *peak* primary load are dropped due to the longer processing times, causing a decrease in primary CPU utilization.

6.1.3 Blind Isolation

We further use the 48 worker variant (*high*) secondary to evaluate the efficiency of isolation mechanisms.

We evaluate the blind isolation mechanism by reserving 4 and 8 buffer logical-cores, respectively. The insight here is to allow the primary to have a buffer of cores to start new worker threads when load increases.

Fig. 5a and Fig. 5b report our findings in terms of latency degradation and CPU utilization. We find that provisioning 8 idle logical cores is enough to ensure less than 1ms of degradation for the 99th latency percentile of the *high* workload.

6.1.4 Comparison to Alternative Isolation Methods

We next analyze the effectiveness of two common methods of static CPU resource management which are available in most modern OSes: *restricting CPU cores* and *restricting CPU cycles*. Windows provides these mechanisms through the *Job Object* abstraction [3], while Linux does so through the *cgroups* framework [4].

Restricting CPU cores. We successively restrict the secondary to use only 24, 16, and 8 cores of all 48 available logical cores. The primary is guaranteed exclusive and unimpeded access to the remainder, but can also compete for the secondary's cores. Fig. 6a shows the degradation of query response latency for each case.

Fig. 6b shows the overall CPU utilization breakdown when the bully is restricted to a subset of cores. When IndexServe is under average load, the secondary can claim up to 33% of the CPU time. While this is an important gain, the servers need to be provisioned for *peak* load, thereby reducing the subset of cores allocated to the secondary to 8 cores. With IndexServe under *peak* load, the secondary can only use up to 17% of the CPU time.

Restricting CPU cycles. We successively restrict the secondary to 45%, 25% and 5% of the overall CPU time.

Fig. 7a reports the measured degradation of latency, and Fig. 7c shows the percentage of queries that were dropped (because of increased processing times). Giving the bully even as little as 5% of CPU time still produces degradation. Furthermore, as opposed to restricting CPU cores, there is always some percentage of queries that get dropped, ranging from 50% to around 1% (in the best case). Fig. 7b shows that using this method less CPU time goes to the secondary.

The main reason this technique yields results worse than restricting CPU cores is that multi-threaded services such as IndexServe launch short-lived worker threads to process incoming requests. If these threads end up being queued for execution instead of being launched right away, it creates a cascading effect which impacts all incoming queries. Despite the secondary not utilizing more than its share of CPU time, IndexServe worker threads get delayed, leading to considerable degradation.

Progress of the secondary. Finally, we analyze the amount of work that the secondary gets done under isolation, as a percentage of the total work done when unrestricted. We report for each IndexServe workload the point where latency degradation was lowest for that experiment. Blind Isolation allows the secondary to achieve 62% and 25% of the work it did unrestricted, for 2,000 and 4,000 QPS, respectively. Restricting CPU cores yields a more modest 45% and a similar 30%, respectively. Restricting CPU cycles fares worst, yielding only 9% in both cases.

6.2 Cluster Experiments

We next look at how *PerfIso* performs in a production cluster. We evaluate at an approximation of peak load to stress the system, attempting to impact tail latency. This makes *PerfIso* throttle the secondary more aggressively to accommodate the primary.

As described in Fig. 3, requests arrive at one of many top-level aggregator (TLA) machines, which forwards the request in a round-robin fashion to one of the two rows of IndexServe machines (each row holds a partitioned copy of the search index). The TLA chooses an IndexServe machine from the row to act as the mid-level aggregator (MLA) for a particular request. The MLA queries all the other IndexServe instances in its row (including the local one), aggregates all results, and formulates the final query response.

We run each experiment 8 times, and measure query latency at a) each server, b) at each layer, and c) end-to-end. Fig. 9a reports the baseline query response latency, averaged across IndexServe machines, across MLAs, and across TLAs. The HDFS client takes up to 5% of total CPU time.

We start our CPU bully and configure *PerfIso* on each IndexServe machine for blind isolation in the same manner as the singlebox runs. Fig. 9b shows the query response latency for this CPU-bound workload. Compared to the baseline, the 99th percentile reported by IndexServe, MLA, and TLA instances increases by at most 0.8, 0.4, and 1.1 milliseconds, respectively.

We configure *PerfIso* to throttle disk I/O to either 100MB/s, or 20 IOPS/s, for a given operation data chunk size of 8KB. Fig. 9c shows the query response latency for this Disk-bound workload. Compared to the baseline, the 99th percentile reported by IndexServe, MLA, and TLA instances increases by at most 0.8, 1.2, and 1.1 milliseconds, respectively.

Progress results with larger cluster. Finally, we show production results for a cluster of 650 IndexServe machines processing live user queries while colocated with a large batch job executing the training phase of a machine-learning computation. Fig. 10 shows key performance metrics: load in QPS, 99th percentile latency of query response times measured at the TLA, and server CPU utilization averaged across all machines. Importantly, CPU utilization averages 70% over 1 hour.

6.3 Takeaways

We now present a sum-up of our evaluation, referring to the objectives established in Section 5.1:

1. Fig. 8a shows that a CPU-bound batch job can importantly affect the 99th percentile query latency of the primary, reaching up to 29× degradation. Fig. 4a shows that even a mildly CPU-intensive job can cause interference and can increase tail latency by up to 42%.

2. Fig. 8a shows that *blind isolation* successfully protects tail latency under medium load (2,000 QPS), and Fig. 4a shows that this holds for *peak* loads (4,000 QPS) as well. In the latter case, the 99th percentile is within 1 ms of the standalone case. Fig. 9 reports the results for 8 runs of an approximated *peak* load (4,000 QPS) on a production cluster, showing that *PerfIso* successfully protects tail latency. The query response latency tail for CPU and Disk-bound jobs (Figs. 9b and 9c) is within 1.2 milliseconds of the standalone case (Fig. 9a). Fig. 10 shows that *blind isolation* raises CPU utilization to 70% through colocation over the course of 1 hour on a 650-machine IndexServe cluster.

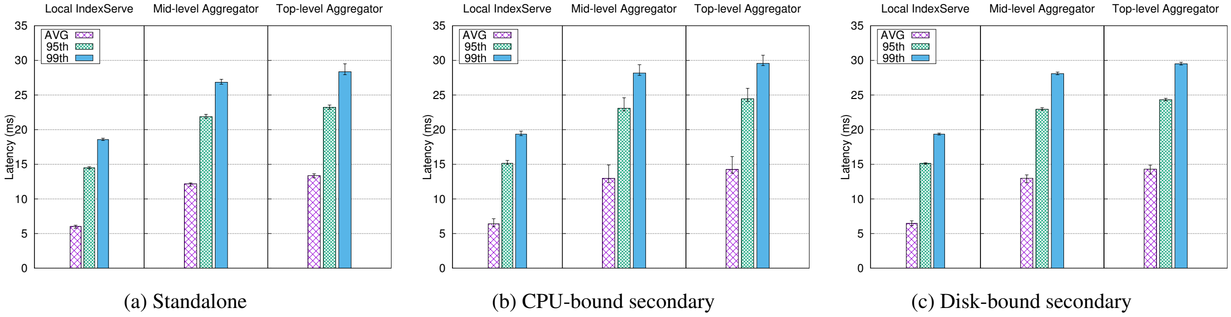
3. Fig. 8 reports a full comparison of all our evaluated techniques, showing that *blind isolation* and *cpu cores* both protect tail latency. However *blind isolation* manages to reduce idle cpu time by a further 13% compared to *cpu cores*, and allows the secondary to perform 17% more work. *CPU cycles* fails to protect tail latency.

We conclude that *PerfIso* successfully protects tail latency across all IndexServe machines, ultimately preserving the end-to-end SLOs.

7 Related Work

Many existing solutions propose colocation to increase data center utilization, but rely on information about the primary tenant's SLO and workload, or on specific hardware support. The complexity and performance characteristics (e.g., tail latency requirements and bursty nature) of primary tenants pushed our design into a different direction, adopting a black-box model with few assumptions on hardware for large-scale deployment.

MS Manners [12] provides CPU-level performance isolation on single-core machines by restricting the CPU



(a) Standalone

(b) CPU-bound secondary

(c) Disk-bound secondary

Figure 9: Latency values for IndexServe running on a production cluster: Fig. 9a shows the baseline, Figs. 9b and 9c show the result of colocation with CPU-bound and Disk-bound secondary tenants, respectively.

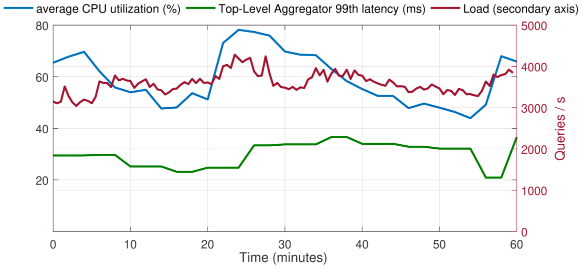


Figure 10: Production results for a cluster of 650 IndexServe machines colocated with a secondary running a machine learning training computation over 1 hour.

cycles available to the secondary, based on job progress information. Our experimental evaluation shows that managing CPU cycles per tenant severely impacts tail latency, and we manage the number of CPU cores of the secondary tenants dynamically instead.

TEMM [38] proposes a throttling-based management technique to configure CPU duty cycles and DVFS settings to meet SLOs. TEMM requires the latency SLOs of the primary, and assumes that the bottleneck is either the last-layer caches or off-chip bandwidth, whereas any of several other resources can be the bottleneck.

MIMP [36] considers tenants in a virtualized environment. It defines a new priority in the hypervisor scheduler to meet the performance requirements of the primary tenant, focusing on CPU-level interference.

Quasar [11] is a cluster manager that reduces resource over-provisioning and increases utilization while meeting quality of service constraints. It uses profiling information and collaborative filtering to infer the tenants' resource requirements. It additionally monitors service performance and adjusts allocations when target latencies are not met. In contrast, *PerfIso*, ensures some slack is always available to accommodate the bursty demands of the primary tenant without assuming explicit server-level performance requirements or tail latency targets.

Heracles [20] uses a feedback mechanism to adjust the secondary tenants' resources based on the tail latency of the primary tenant. Heracles needs the latency SLOs of the primary and exploits hardware mechanisms such as Intel's Cache Allocation Technology [2]. *Heracles* is complementary to this work since the knowledge of primary tenant SLOs and availability of specific hardware mechanisms limits wider deployment.

Jail [28] is Google's cache partitioning performance isolation mechanism. It incorporates Intel CMT and CAT [2] support into Linux cgroups, but allows only static partitioning of resources. In contrast our experiments show dynamic isolation techniques (such as *blind isolation*) provide more resources to the secondary while protecting the primary's tail latency. Additionally, data center machines span multiple hardware generations, not all of which supporting CAT.

RubikColoc [16] configures per-core DVFS to compensate for the overhead of multiplexing primary and secondary tenants on the same core. RubikColoc needs server-level latency SLOs and per-core DVFS support.

Elfen [34] provides CPU-level performance isolation for primary and secondary tenants running on the same core using Simultaneous Multi-Threading (SMT) technology available on modern CPUs. Effectively, the secondary is colocated with the primary on the same physical core only when the primary's measured performance is within its SLO. Elfen needs latency SLOs for the primary, OS support to query SMT-to-process mappings, and application-level instrumentation for the secondary.

BatchDB [23] is a database system that handles both OLTP and OLAP queries, providing good performance isolation for the former. Our approach instead focuses on the scenario where the tenants are distinct applications.

Leverich et al. [18] advocate using colocation to improve cluster throughput-per-TCO, and identify queuing delay, scheduling delay, and worker-thread load imbalance as the challenges in providing service-level QoS. They propose better cluster provisioning and custom OS schedulers to mitigate tenant interference. However, given the complexity of large commercial services, we cannot make changes, neither at the hardware nor kernel level, which is why we adopt a "black-box" model.

CPI^2 [37] is a performance isolation mechanism which uses Clock Per Instruction (CPI) data to build probabilistic distribution models and to find stragglers (victims of resource interference). CPI^2 identifies the antagonists of latency-sensitive tasks by matching CPI patterns, and restricts their CPU cycle-share. We show that restricting CPU cores is significantly more effective in protecting primary tail latency.

HipsterCo [26] is a task scheduler for latency-sensitive workloads running on heterogeneous multi-core systems. HipsterCo colocates batch jobs with latency-sensitive workloads to increase resource utilization. HipsterCo uses reinforcement learning to build the CPU and DVFS configuration required to meet target SLOs. All remaining CPU cores are allocated to batch jobs. However, HipsterCo requires performance feedback from latency-sensitive workloads which is not available in our environment. Furthermore, we argue that for complex commercial latency-sensitive services, some buffer cores are required to prevent performance degradation.

Bubble-Up [24] and *Bubble-Flux* [33] focus on memory bandwidth and last-level cache interference as the main actors in colocation performance degradation. The former proposes a static profiling technique to accurately estimate the expected degradation, while the later uses an online approach, both assuming live performance information from the latency-sensitive service is available.

Zhang et al. [39] use historical resource utilization data and disk re-imaging patterns of tenants in task scheduling and data placement. The primary has resource-priority, meaning that a load-surge kills off secondary tasks. Thus, the scheduling algorithm places secondaries as to minimize the likelihood of termination. *Misra et al.* [25] improved upon this work by proposing a scalable distributed file system design which maximizes data availability for secondary tenants. Due to the highly spiking and unpredictable nature of the primary services we target, relying on historical data is insufficient to insure that performance guarantees are met.

Pisces [30] achieves fairness and per-tenant performance isolation in shared key-value storage. Pisces uses deficit-weighted-round-robin (DWRR) to schedule requests at server-level, thus mediating resource contention. In our case secondary tasks are batch jobs, and therefore do not lend themselves to request scheduling, so we only employ DWRR for I/O throttling.

2DFQ [22] proposes a new weighted fair queuing algorithm to ensure fairness for multi-tenant services that use thread pools inside a single process. This technique benefits the primary tenants and could reduce the burstiness of their execution, which complements *PerfIso* and potentially reduces the number of buffer cores required.

Alizadeh et. al [6] propose HULL — a system which leaves 'bandwidth headroom' to mitigate the problem of packet queuing in low-latency networked systems. This is similar in spirit to our non-work-conserving resource management approach, but focuses on avoiding network congestion rather than performance isolation.

8 Conclusions

Machines hosting large commercial latency-sensitive services are often underutilized because important services are provisioned for peak load as to meet business availably and fault-tolerance constraints.

Colocating batch jobs with latency-sensitive services is an important way of increasing utilization and data center efficiency, but comes with key challenges. Large latency-sensitive services are complex, and follow a layered architecture and therefore require short tail latencies. The diversity of commercial systems prevents us from using explicit knowledge of their latency targets, motivating us to adopt an approach in which we assume limited information about the primary tenant.

This paper presents the design and implementation of *PerfIso*, a performance isolation framework that makes use of idle resources to run batch jobs without affecting the primary tenant. It uses *CPU blind isolation* to meet the requirements of commercial large latency-sensitive services. The key insight is ensuring that the primary tenant always has idle cores available to accommodate its bursty workload, while allowing the secondary tenant to make progress.

We evaluate *PerfIso* experimentally on a single machine and on a cluster of machines using Bing IndexServe as the primary workload. We compare existing CPU isolation techniques (such as rate limiting and static core affinitization) to our approach, and we find that under the latter the 99^{th} percentile of the tail latency values remain largely unchanged compared to running standalone. *PerfIso* allows compute-intensive batch jobs to use up to 47% of CPU cycles for off-peak loads which would have otherwise remained idle.

Acknowledgements We thank our anonymous reviewers for their valuable feedback and advice. We thank our colleagues in Microsoft Windows: Danyu Zhu, Kai Hsu, Deepu Thomas, Mehmet Iyigun, Milos Kralj and Eugene Bak, as well as Ricardo Bianchini (Microsoft Research), Íñigo Goiri (Microsoft Research), Willy Zwaenepoel (EPFL), Marcus Fountoura (Microsoft Azure), Ramashis Das (Microsoft Bing), Kukjin Lee (Microsoft Research).

References

[1] Hadoop. http://hadoop.apache.org.

[2] Intel CAT. https://www.intel.com/content/www/us/en/communications/cache-monitoring-cache-allocation-technologies.html.

[3] Windows Job Objects. https://msdn.microsoft.com/en-us/library/windows/desktop/hh684161(v=vs.85).aspx.

[4] Cgroups, 2014. http://en.wikipedia.org/wiki/Cgroups.

[5] DiskSPD, 2017. https://github.com/Microsoft/diskspd.

[6] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 19–19.

[7] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark SQL: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 1383–1394.

[8] Barroso, L. A., Clidaras, J., and Hölzle, U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture 8*, 3 (2013), 1–154.

[9] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36*, 4 (2015).

[10] Dean, J., and Barroso, L. A. The tail at scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[11] Delimitrou, C., and Kozyrakis, C. Quasar: resource-efficient and QoS-aware cluster management. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 127–144.

[12] Douceur, J. R., and Bolosky, W. J. Progress-based regulation of low-importance processes. In *In Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (1999), ACM Press, pp. 247–260.

[13] Fedorova, A., Seltzer, M., and Smith, M. D. A non-work-conserving operating system scheduler for SMT processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA* (2006), vol. 33, pp. 10–17.

[14] Isard, M. Autopilot: Automatic data center management. *ACM SIGOPS Operating Systems Review 41*, 2 (Apr. 2007), 60–67.

[15] Jeon, M., He, Y., Kim, H., Elnikety, S., Rixner, S., and Cox, A. L. TPC: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ACM, pp. 129–141.

[16] Kasture, H., Bartolini, D. B., Beckmann, N., and Sanchez, D. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015), ACM, pp. 598–610.

[17] Kim, S., He, Y., Hwang, S.-w., Elnikety, S., and Choi, S. Delayed-Dynamic-Selective (DDS) prediction for reducing extreme tail latency in web search. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining* (2015), ACM, pp. 7–16.

[18] Leverich, J., and Kozyrakis, C. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 4.

[19] Li, T., Baumberger, D., and Hahn, S. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 65–74.

[20] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Improving resource efficiency at scale with Heracles. *ACM Transactions on Computer Systems (TOCS) 34*, 2 (2016), 6.

[21] Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V., and Fedorova, A. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 1.

[22] Mace, J., Bodik, P., Musuvathi, M., Fonseca, R., and Varadarajan, K. 2dfq: Two-dimensional fair queuing for multi-tenant cloud services. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 144–159.

[23] Makreshanski, D., Giceva, J., Barthels, C., and Alonso, G. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 37–50.

[24] Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M. L. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture* (2011), ACM, pp. 248–259.

[25] Misra, P. A., Goiri, I., Kace, J., and Bianchini, R. Scaling distributed file systems in resource-harvesting datacenters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 799–811.

[26] Nishtala, R., Carpenter, P., Petrucci, V., and Martorell, X. Hipster: Hybrid task manager for latency-critical cloud workloads. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on* (2017), IEEE, pp. 409–420.

[27] Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., Chun, B.-G., and ICSI, V. Making sense of performance in data analytics frameworks. In *NSDI* (2015), vol. 15, pp. 293–307.

[28] Rohit, J., and David, L. CAT at scale: Deploying cache isolation in a mixed workload environment. LinuxCon + ContainerCon North America, August 2016.

[29] Schurman, E., and Brutlag, J. Performance related changes and their user impact. In *velocity web performance and operations conference* (2009).

[30] Shue, D., Freedman, M. J., and Shaikh, A. Performance isolation and fairness for multi-tenant cloud storage. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 349–362.

[31] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. Apache hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), ACM, p. 5.

[32] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 18.

[33] YANG, H., BRESLOW, A., MARS, J., AND TANG, L. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 607–618.

[34] YANG, X., BLACKBURN, S. M., AND MCKINLEY, K. S. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *USENIX Annual Technical Conference* (2016), pp. 309–322.

[35] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., ET AL. Apache Spark: A unified engine for big data processing. *Communications of the ACM 59*, 11 (2016), 56–65.

[36] ZHANG, W., RAJASEKARAN, S., DUAN, S., WOOD, T., AND ZHUY, M. Minimizing interference and maximizing progress for Hadoop virtual machines. *ACM SIGMETRICS Performance Evaluation Review 42*, 4 (2015), 62–71.

[37] ZHANG, X., TUNE, E., HAGMANN, R., JNAGAL, R., GOKHALE, V., AND WILKES, J. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 379–391.

[38] ZHANG, X., ZHONG, R., DWARKADAS, S., AND SHEN, K. A flexible framework for throttling-enabled multicore management (TEMM). In *Parallel Processing (ICPP), 2012 41st International Conference on* (2012), IEEE, pp. 389–398.

[39] ZHANG, Y., PREKAS, G., FUMAROLA, G. M., FONTOURA, M., GOIRI, I., AND BIANCHINI, R. History-based harvesting of spare cycles and storage in large-scale datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 755–770.

On the diversity of cluster workloads and its impact on research results

George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson
Carnegie Mellon University
Elisabeth Baseman, Nathan DeBardeleben
Los Alamos National Laboratory

Abstract

Six years ago, Google released an invaluable set of scheduler logs which has already been used in more than 450 publications. We find that the scarcity of other data sources, however, is leading researchers to overfit their work to Google's dataset characteristics. We demonstrate this overfitting by introducing four new traces from two private and two High Performance Computing (HPC) clusters. Our analysis shows that the private cluster workloads, consisting of data analytics jobs expected to be more closely related to the Google workload, display more similarity to the HPC cluster workloads. This observation suggests that additional traces should be considered when evaluating the generality of new research.

To aid the community in moving forward, we release the four analyzed traces, including: the longest publicly available trace spanning *all* 61 months of an HPC cluster's lifetime and a trace from a 300,000-core HPC cluster, the largest cluster with a publicly available trace. We present an analysis of the private and HPC cluster traces that spans job characteristics, workload heterogeneity, resource utilization, and failure rates. We contrast our findings with the Google trace characteristics and identify affected work in the literature. Finally, we demonstrate the importance of dataset plurality and diversity by evaluating the performance of a job runtime predictor using all four of our traces and the Google trace.

1 Introduction

Despite intense activity in the areas of cloud and job scheduling research, publicly available cluster workload datasets remain scarce. The three major dataset sources today are: the Google cluster trace [58] collected in 2011, the Parallel Workload Archive [19] of High Performance Computing (HPC) traces collected since 1993, and the SWIM traces released in 2011 [10]. Of these, the Google trace has been used in more than 450 publications making it the most popular trace by far. Unfortunately, this 29-day trace is often the only one used to evaluate new research. By contrasting its characteristics with newer traces from different environments, we have found that the Google trace alone is insufficient to accurately prove the generality of a new technique.

Our goal is to uncover overfitting of prior work to the characteristics of the Google trace. To achieve this, our first contribution is to introduce four new traces: two from the private cloud of Two Sigma, a hedge fund, and two from HPC clusters located at the Los Alamos National Laboratory (LANL). Our Two Sigma traces are the longest, non-academic private cluster traces to date, spanning 9 months and more than 3 million jobs. The two HPC traces we introduce are also unique. The first trace spans the entire 5-year lifetime of a general-purpose HPC cluster, making it the longest public trace to date, while also exhibiting shorter jobs than existing public HPC traces. The second trace originates from the 300,000-core current flagship supercomputer at LANL, making it the largest cluster with a public trace, to our knowledge. We introduce all four traces, and the environments where they were collected, in Section 2.

Our second contribution is an analysis examining the generality of workload characteristics derived from the Google trace, when our four new traces are considered. Overall, we find that the private Two Sigma cluster workloads display similar characteristics to HPC, despite consisting of data analytics jobs that more closely resemble the Google workload. Table 1 summarizes all our findings. For those characteristics where the Google workload is an outlier, we have surveyed the literature and list affected prior work. In total, we surveyed 450 papers that reference the Google trace study [41] to identify popular workload assumptions, and we constrast them to the Two Sigma and LANL workloads to detect violations. We group our findings into four categories: job characteristics (Section 3), workload heterogeneity (Section 4), resource utilization (Section 5), and failure analysis (Section 6).

Our findings suggest that evaluating new research using the Google trace alone is insufficient to guarantee generality. To aid the community in moving forward, our third contribution is to publicly release the four traces introduced and analyzed in this paper. We further present a case study on the importance of dataset plurality and diversity when evaluating new research. For our demonstration we use JVuPredict, the job runtime predictor of the JamaisVu scheduling system [51]. Originally, JVuPredict was evaluated using only the Google trace [51]. Evaluating its performance with our four new traces, however, helped us identify features that make it easier to detect related and recurring jobs with predictable behavior. This enabled us to quantify the importance of individual trace fields in runtime prediction.

Section	Characteristic	Google	Two Sigma	Mustang	OpenTrinity
Job Characteristics (§3)	Majority of jobs are small	✔	✘	✘	✘
	Majority of jobs are short	✔	✘	✘	✘
Workload Heterogeneity (§4)	Diurnal patterns in job submissions	✘	✔	✔	✔
	High job submission rate	✔	✔	✘	✘
Resource Utilization (§5)	Resource over-commitment	✔	✘	✘	✘
	Sub-second job inter-arrival periods	✔	✔	✔	✔
	User request variability	✘	✔	✔	✔
Failure Analysis (§6)	High fraction of unsuccessful job outcomes	✔	✔	✘	✔
	Jobs with unsuccessful outcomes consume significant fraction of resources	✔	✔	✘	✘
	Longer/larger jobs often terminate unsuccessfully	✔	✘	✘	✘

Table 1: Summary of the characteristics of each trace. Note that the Google workload appears to be an outlier.

We describe our findings in Section 7.

Finally, we briefly discuss the importance of trace length in accurately representing a cluster's workload in Section 8. We list related work studying cluster traces in Section 9, before concluding.

2 Dataset information

We introduce four sets of job scheduler logs that were collected from a general-purpose cluster and a cutting-edge supercomputer at LANL, and across two clusters of Two Sigma, a hedge fund. The following subsections describe each dataset in more detail, and the hardware configuration of each cluster is shown in Table 2.

Users typically interact with the cluster scheduler by submitting commands that spawn multiple processes, or *tasks*, distributed across cluster nodes to perform a specific computation. Each such command is considered to be a *job* and users often compose scripts that generate more complex, multi-job schedules. In HPC clusters, where resources are allocated at the granularity of physical nodes similar to Emulab [4, 16, 27, 57], tasks from different jobs are never scheduled on the same node. This is not necessarily true in private clusters like Two Sigma.

2.1 Two Sigma clusters

The private workload traces we introduce originate from two datacenters of Two Sigma, a hedge fund firm. The workload consists of data analytics jobs processing financial data. A fraction of these jobs are handled by a Spark [49] installation, while the rest are serviced by home-grown data analytics frameworks. The dataset spans 9 months of the two datacenters' operation starting in January 2016, covering a total of 1313 identical compute nodes with 31512 CPU cores and 328TB RAM. The logs contain 3.2 million jobs and 78.5 million tasks, collected by an internally-developed job scheduler running on top of Mesos [28]. Because both datacenters experience the same workload and consist of homogeneous nodes, we collectively refer to both data sources as the *TwoSigma* trace, and analyze them together.

Platform	Nodes	CPUs	RAM	Length
LANL Trinity	9408	32	128GB	3 months
LANL Mustang	1600	24	64GB	5 years
TwoSigma A	872	24	256GB	9 months
TwoSigma B	441	24	256GB	
Google B	6732	0.50*	0.50*	29 days
Google B	3863	0.50*	0.25*	
Google B	1001	0.50*	0.75*	
Google C	795	1.00*	1.00*	
Google A	126	0.25*	0.25*	
Google B	52	0.50*	0.12*	
Google B	5	0.50*	0.03*	
Google B	5	0.50*	0.97*	
Google C	3	1.00*	0.50*	
Google B	1	0.50*	0.06*	

Table 2: Hardware characteristics of the clusters analyzed in this paper. For the Google trace [41], (*) signifies a resource has been normalized to the largest node.

We expect this workload to resemble the Google cluster more closely than the HPC clusters, where long-running, compute-intensive, and tightly-coupled scientific jobs are the norm. First, unlike LANL, job runtime is not budgeted strictly; users of the hedge fund clusters do not have to specify a time limit when submitting a job. Second, users can allocate individual cores, as opposed to entire physical nodes allocated at LANL. Collected data include: timestamps for job stages from submission to termination, job properties such as size and owner, and the job's return status.

2.2 LANL Mustang cluster

Mustang was an HPC cluster used for *capacity computing* at LANL from 2011 to 2016. Capacity clusters such as Mustang are architected as cost-effective, general-purpose resources for a large number of users. Mustang was largely used by scientists, engineers, and software developers at LANL and it was allocated to these users at the granularity of physical nodes. The cluster consisted of 1600 identical compute nodes, with a total of 38400 AMD Opteron 6176 2.3GHz cores and 102TB RAM.

Our Mustang dataset covers the entire 61 months of the machine's operation from October 2011 to November 2016, which makes this the longest publicly available cluster trace to date. The Mustang trace is also unique because its jobs are shorter than those in existing HPC traces. Overall, it consists of 2.1 million multi-node jobs submitted by 565 users and collected by SLURM [45], an open-source cluster resource manager. The fields available in the trace are similar to those in the TwoSigma trace, with the addition of a time budget field per job, that if exceeded causes the job to be killed.

2.3 LANL Trinity supercomputer

In 2018, Trinity is the largest supercomputer at LANL and it is used for *capability computing*. Capability clusters are a large-scale, high-demand resource introducing novel hardware technologies that aid in achieving crucial computing milestones, such as higher-resolution climate and astrophysics models. Trinity's hardware was stood up in two pre-production phases before being put into full production use and our trace was collected before the second phase completed. At the time of data collection, Trinity consisted of 9408 identical compute nodes, a total of 301056 Intel Xeon E5-2698v3 2.3GHz cores and 1.2PB RAM, making this the largest cluster with a publicly available trace by number of CPU cores.

Our Trinity dataset covers 3 months from February to April 2017. During that time, Trinity was operating in OpenScience mode, i.e., the machine was undergoing beta testing and was available to a wider number of users than it is expected to have after it receives its final security classification. We note that OpenScience workloads are representative of a capability supercomputer's workload, as they occur roughly every 18 months when a new machine is introduced, or before an older one is decommissioned. The dataset, which we will henceforth refer to as *OpenTrinity*, consists of 25237 multi-node jobs issued by 88 users and collected by MOAB [1], an open-source cluster scheduling system. The information available in the trace is the same as that in the Mustang trace.

2.4 Google cluster

In 2012, Google released a trace of jobs that ran in one of their compute clusters [41]. It is a 29-day trace consisting of 672074 jobs and 48 million tasks, some of which were issued through the MapReduce framework, and ran on 12583 heterogeneous nodes in May 2011. The workload consists of both long-running services and batch jobs [55]. Google has not released the exact hardware specifications of each cluster node. Instead, as shown in Table 2, nodes are presented through anonymized platform names representing machines with different combinations of microarchitectures and chipsets [58]. Note that the number of CPU cores and RAM for each node in the trace have been normalized to the most powerful node in the cluster. In our analysis, we estimate the total number of cores in the Google cluster to be 106544. We derive this number by assuming that the most popular node type (Google B with 0.5 CPU cores) is a dual-socket server, carrying quad-core AMD Opteron Barchelona CPUs that Google allegedly used in their datacenters at the time [26]. Unlike previous workloads, jobs can be allocated fractions of a CPU core [46].

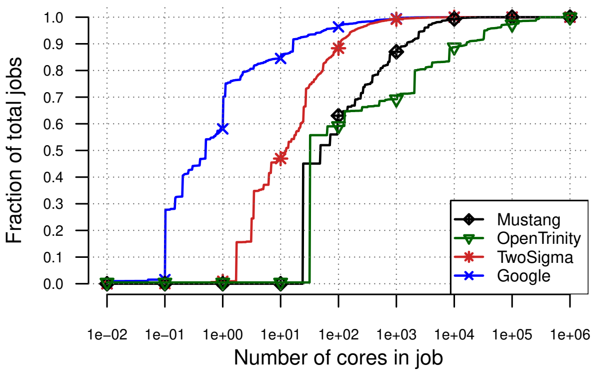


Figure 1: CDF of job sizes based on allocated CPU cores.

3 Job characteristics

Many instances of prior work in the literature rely on the assumption of heavy-tailed distributions to describe the size and duration of individual jobs [2, 8, 13, 14, 40, 50]. In the LANL and TwoSigma workloads these tails appear significantly lighter.

Observation 1: *On average, jobs in the TwoSigma and LANL traces request 3 - 406 times more CPU cores than jobs in the Google trace. Job sizes in the LANL traces are more uniformly distributed.*

Figure 1 shows the Cumulative Distribution Functions (CDFs) of job requests for CPU cores across all traces, with the x-axis in logarithmic scale. We find that the 90% of smallest jobs in the Google trace request 16 CPU cores or fewer. The same fraction of TwoSigma jobs request 108 cores, and 1-16K cores in the LANL traces. Very large jobs are also more common outside Google. This is unsurprising for the LANL HPC clusters, where allocating thousands of CPU cores to a single job is not uncommon, as the clusters' primary use is to run massively parallel scientific applications. It is interesting to note, however, that while the TwoSigma clusters contain fewer cores than the other clusters we examine (3 times fewer

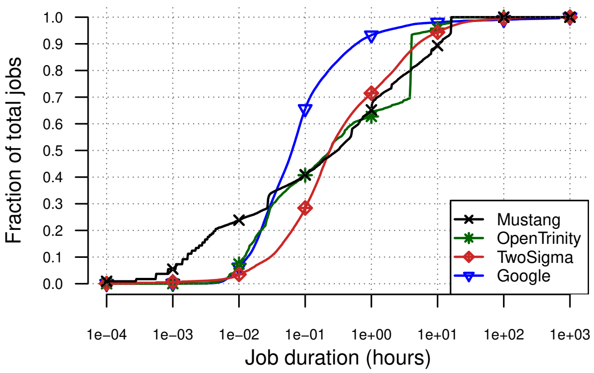


Figure 2: CDF of the durations of individual jobs.

than the Google cluster), its median job is more than an order of magnitude larger than a job in the Google trace. An analysis of allocated memory yields similar trends.

Observation 2: *The median job in the Google trace is 4-5 times shorter than in the LANL or TwoSigma traces. The longest 1% of jobs in the Google trace, however, are 2-6 times longer than the same fraction of jobs in the LANL and TwoSigma traces.*

Figure 2 shows the CDFs of job durations for all traces. We find that in the Google trace, 80% of jobs last less than 12 minutes *each*. In the LANL and TwoSigma traces jobs are at least an order of magnitude longer. In TwoSigma, the same fraction of jobs last up to 2 hours and in LANL, they last up to 3 hours for Mustang and 6 hours for OpenTrinity. Surprisingly, the tail end of the distribution is slightly shorter for the LANL clusters than for the Google and TwoSigma clusters. The longest job is 16 hours on Mustang, 32 hours in OpenTrinity, 200 hours in TwoSigma, and at least 29 days in Google (the duration of the trace). For LANL, this is due to hard limits causing jobs to be indiscriminately killed. For Google, the distribution's long tail is likely attributed to long-running services.

Implications. These observations impact the immediate applicability of job scheduling approaches whose efficiency relies on the assumption that the vast majority of jobs' durations are in the order of minutes, and job sizes are insignificant compared to the size of the cluster. For example, Ananthanarayanan et al. [2] propose to mitigate the effect of stragglers by duplicating tasks of smaller jobs. This is an effective approach for Internet service workloads (Microsoft and Facebook are represented in the paper) because the vast majority of jobs can benefit from it, without significantly increasing the overall cluster utilization. For the Google trace, for example, 90% of jobs request less than 0.01% of the cluster each, so duplicating them only slightly increases cluster utilization. At the same time, 25-55% of jobs in the LANL and TwoSigma traces *each* request *more than* 0.1% of the cluster's cores, decreasing the efficiency of the approach and suggesting replication should be used judiciously. This does not consider that LANL tasks are also tightly-coupled and the entire job has to be duplicated.

Another example is the work by Delgado et al. [14], which improves the efficiency of distributed schedulers for short jobs by dedicating them a fraction of the cluster. This partition ranges from 2% for Yahoo and Facebook traces, to 17% for the Google trace where jobs are significantly longer, to avoid increasing job service times. For the TwoSigma and LANL traces we have shown that jobs are even longer than for the Google trace (Figure 2), so larger partitions will likely be necessary to achieve similar efficiency. At the same time, jobs running in the TwoSigma and LANL clusters are also larger (Figure 1), so service times for long jobs are expected to increase unless the partition is shrunk. Other examples of work that is likely affected include task migration of short and small jobs [50] and hybrid scheduling aimed on improving head-of-line blocking for short jobs [13].

4 Workload heterogeneity

Another common assumption about cloud workloads is that they are characterized by heterogeneity in terms of resources available to jobs, and job interarrival times [7, 23, 31, 46, 56]. The private and HPC clusters we study, however, consist of homogeneous hardware (see Table 2) and user activity follows well-defined diurnal patterns, even though the rate of scheduling requests varies significantly across clusters.

Observation 3: *Diurnal patterns are universal. Clusters received more scheduling requests and smaller jobs at daytime, with minor deviations for the Google trace.*

In Figure 3 we show the number of job scheduling requests for every hour of the day. We choose to show metrics for the median day surrounded by the other two quartiles because the high variation across days causes the averages to be unrepresentative of the majority of days (see Section 8). Overall, diurnal patterns are evident in every trace and user activity is concentrated at daytime (7AM to 7PM), similar to prior work [38]. An exception to this is the Google trace, which is most active from midnight to 4AM, presumably due to batch jobs leveraging the available resources.

Sizes of submitted jobs are also correlated with the time of day. We find that longer, larger jobs in the LANL traces are typically scheduled during the night, while shorter, smaller jobs tend to be scheduled during the day. The reverse is true for the Google trace, which prompts our earlier assumption on nightly batch jobs. Long, large jobs are also scheduled at daytime in the TwoSigma clusters, despite having a diurnal pattern similar to LANL

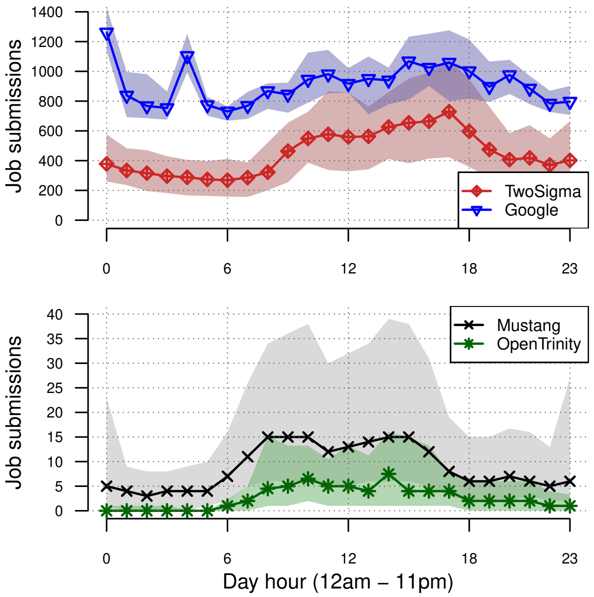


Figure 3: Hourly job submission rates for a given day. The lines represent the median, while the shaded region shows the distance between the 25th and 75th percentiles.

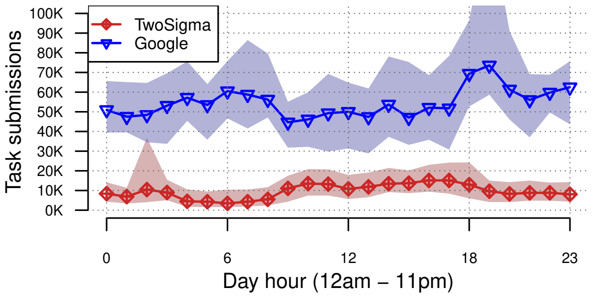


Figure 4: Hourly task placement requests for a given day. The lines represent the median, while the shaded region shows the distance between the 25th and 75th percentiles.

clusters. This is likely due to TwoSigma's workload consisting of financial data analysis, which bears a dependence on stock market hours.

Observation 4: *Scheduling request rates differ by up to 3 orders of magnitude across clusters. Sub-second scheduling decisions seem necessary in order to keep up with the workload.*

One more thing to take away from Figure 3 is that the rate of scheduling requests can differ significantly across clusters. For the Google and TwoSigma traces, hundreds to thousands of jobs are submitted every hour. On the other hand, LANL schedulers never receive more than 40 requests on any given hour. This could be related to the workload or the number of users in the system, as the Google cluster serves 2 times as many user IDs as the Mustang cluster and 9 times as many as OpenTrinity.

Implications: Previous work such as Omega [46] and ClusterFQ [56] propose distributed scheduling designs especially applicable to heterogeneous clusters. This does not seem to be an issue for environments such as LANL and TwoSigma, which intentionally architect homogeneous clusters to lower performance optimization and administration costs.

As cluster sizes increase, so does the rate of scheduling requests, urging us to reexamine prior work. Quincy [31] represents scheduling as a Min-Cost Max-Flow (MCMF) optimization problem over a task-node graph and continuously refines task placement. The complexity of this approach, however, becomes a drawback for large-scale clusters such as the ones we study. Gog et al. [23] find that Quincy requires 66 seconds (on average) to converge to a placement decision in a 10,000-node cluster. The Google and LANL clusters we study already operate on that scale (Table 2). We have shown in Figure 3 that the average frequency of job submissions in the LANL traces is one job every 90 seconds, which implies that this scheduling latency may work, but this will not be the case for long. Trinity is currently operating with 19,000 nodes and, under the DoE's Exascale Computing Project [39], 25 times larger machines are planned within the next 5 years. Note that when discussing scheduling so far we refer to *jobs*, since HPC jobs have a gang scheduling requirement. Placement algorithms such as Quincy, however, focus on *task* placement.

An improvement to Quincy is Firmament [23], a centralized scheduler employing a generalized approach based on a combination of MCMF optimization techniques to achieve sub-second task placement latency on average. As Figure 4 shows, sub-second latency is paramount, since the rate of task placement requests in the Google and TwoSigma traces can be as high as 100K requests per hour, i.e. one task every 36ms. Firmament's placement latency, however, increases to several seconds as cluster utilization increases. For the TwoSigma and Google traces this can be problematic.

5 Resource utilization

A well-known motivation for the cloud has been resource consolidation, with the intention of reducing equipment ownership costs. An equally well-known property of the cloud, however, is that its resources remain underutilized [6, 15, 35, 36, 41]. This is mainly due to a disparity between user resource requests and actual resource usage, which recent research efforts try to alleviate through workload characterization and aggressive consolidation [15, 33, 34]. Our analysis finds that user resource requests in the LANL and TwoSigma traces are characterized by higher variability than in the Google trace. We also look into job inter-arrival times and how they are

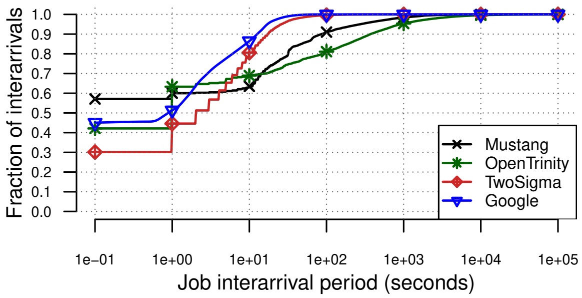


Figure 5: CDF of job interarrival times.

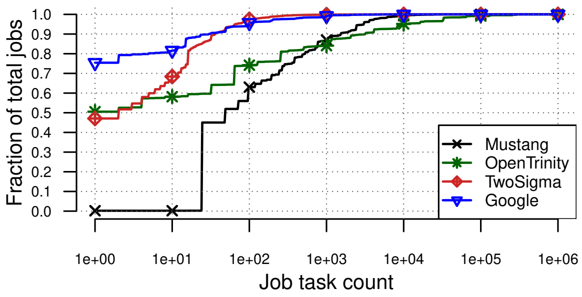


Figure 6: CDF of the number of tasks per job.

approximated when evaluating new research.

Observation 5: *Unlike the Google cluster, none of the other clusters we examine overcommit resources.*

Overall, we find that the fraction of CPU cores allocated to jobs is stable over time across all the clusters we study. For Google, CPU cores are over provisioned by 10%, while for other clusters unallocated cores range between 2-12%, even though resource overprovisioning is supported by their schedulers. Memory allocation numbers follow a similar trend. Unfortunately, the LANL and TwoSigma traces do not contain information on actual resource utilization. As a result, we can neither confirm, nor contradict results from earlier studies on the imbalance between resource allocation and utilization. What differs between organizations is the motivation for keeping resources utilized or available. For Google [41], Facebook [10], and Twitter [15], there is a tension between the financial incentive of maintaining only the necessary hardware to keep operational costs low and the need to provision for peak demand, which leads to low overall utilization. For LANL, clusters are designed to accommodate a predefined set of applications for a predetermined time period and high utilization is planned as part of efficiently utilizing federal funding. For the TwoSigma clusters, provisioning for peak demand is more important, even if it leads to low overall utilization, since business revenue is heavily tied to the response times of their analytics jobs.

Observation 6: *The majority of job interarrivals periods are sub-second in length.*

Interarrival periods are a crucial parameter of an experimental setup, as they dictate the load on the system under test. Two common configurations are second-granularity [15] or Poisson-distributed interarrivals [29], and we find that neither characterizes interarrivals accurately. In Figure 5 we show the CDFs for job interarrival period lengths. We observe that 44-62% of interarrival periods are sub-second, implying that jobs arrive at a faster rate than previously assumed. Furthermore, our attempts to fit a Poisson distribution on this data have been unsuccessful, as Kolmogorov-Smirnov tests [37] reject the null hypothesis with p-values $< 2.2 \times 10^{-16}$. This result does not account for a scenario where there is an underlying Poisson process with a rate parameter changing over time, but it suggests that caution should be used when a Poisson distribution is assumed.

Another common assumption is that jobs are very rarely big, i.e., made up of multiple tasks [29, 56]. In Figure 6 we show the CDFs for the number of tasks per job across organizations. We observer that 77% of Google jobs are single-task jobs, but the rest of the clusters carry many more multi-task jobs. We note that the TwoSigma distribution approaches that of Google only for larger jobs. This suggests that task placement may be a harder problem outside Google, where single-task jobs are common, exacerbating the evaluation issues we outlined in Section 4 for existing task placement algorithms.

Observation 7: *User resource requests are more variable in the LANL and TwoSigma traces than in the Google trace.*

Resource under-utilization can be alleviated through workload consolidation. To ensure minimal interference, applications are typically profiled and classified according to historical data [15, 33]. Our analysis suggests that this approach is likely to be less successful outside the Internet services world. To quantify variability in user behavior we examine the Coefficient of Variation[1] (CoV) across all requests of individual users. For the Google trace we find that the majority of users issue jobs within 2x of their average request in CPU cores. For the LANL and TwoSigma traces, on the other hand, 60-80% of users can deviate by 2-10x of their average request.

Implications: A number of earlier studies of Google [41], Twitter [15], and Facebook [10] data have highlighted the imbalance between resource allocation and utilization. Google tackles this issue by over-committing resources, but this is not the case for LANL and TwoSigma. Another proposed solution is Quasar [15], a system that consolidates workloads while guaranteeing

[1] The Coefficient of Variation is a unit-less measure of spread, derived by dividing a sample's standard deviation by its mean.

a predefined level of QoS. This is achieved by profiling jobs at submission time and classifying them as one of the previously encountered workloads; misclassifications are detected by inserting probes in the running application. For LANL, this approach would be infeasible. First, jobs cannot be scaled down for profiling, as submitted codes are often carefully configured for the requested allocation size. Second, submitted codes are too complex to be accurately profiled in seconds, and probing them at runtime to detect misclassifications can introduce performance jitter that is prohibitive in tightly-coupled HPC applications. Third, in our LANL traces we often find that users tweak jobs before resubmitting them, as they re-calibrate simulation parameters to achieve a successful run, which is likely to affect classification accuracy. Fourth, resources are carefully reserved for workloads and utilization is high, which makes it hard to provision resources for profiling. For the TwoSigma and Google traces Quasar may be a better fit, however, at the rate of 2.7 jobs per second (Figure 3), 15 seconds of profiling [15] at submission time would result in an expected load of 6 jobs being profiled together. Since Quasar requires 4 parallel and isolated runs to collect sufficient profiling data, we would need resources to run at least 360 VMs concurrently, with guaranteed performance isolation between tham to keep up with the average load. This further assumes the profiling time does not need to be increased beyond 15 seconds. Finally, Quasar [15] was evaluated using multi-second inter-arrival periods, so testing would be necessary to ensure that one order of magnitude more load can be handled (Figure 5), and that it will not increase the profiling cost further.

Another related approach to workload consolidation is provided by TSF [56], a scheduling algorithm that attempts to maximize the number of task slots allocated to each job, without favoring bigger jobs. This ensures that the algorithm remains starvation-free, however it results in significant slowdowns in the runtime of jobs with 100+ tasks, which the authors define as big. This would be prohibitive for LANL, where jobs must be scheduled as a whole, and such "big" jobs are much more prevalent and longer in duration. Other approaches for scheduling and placement assume the availability of resources that may be unavailable in the clusters we study here, and their performance is shown to be reduced in highly-utilized clusters [25, 29].

6 Failure analysis

Job scheduler logs are often analyzed to gain an understanding of job failure characteristics in different environments [9, 17, 21, 22, 43]. This knowledge allows for building more robust systems, which is especially important as we transition to exascale computing systems where failures are expected every few minutes [48], and cloud computing environments built on complex software stacks that increase failure rates [9, 44].

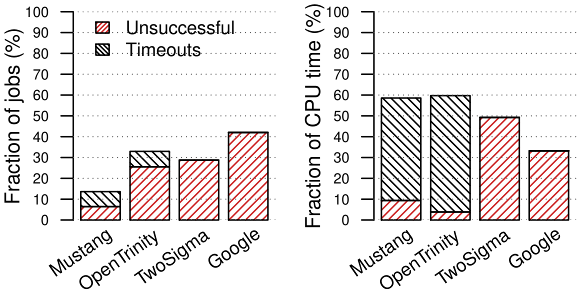


Figure 7: Breakdown of the total number of jobs, as well as CPU time, by job outcome.

Definitions. An important starting point for any failure analysis is defining what constitutes a failure event. Across all traces we consider, we define as *failed jobs* all those that end due to events whose occurrence was not intended by users or system administrators. We do not distinguish failed jobs by their root cause, e.g., software and hardware issues, because this information is not reliably available. There are other job termination states in the traces, in addition to success and failure. For the Google trace, jobs can be killed by users, tasks can be evicted in order to schedule higher-priority ones, or have an unknown exit status. For the LANL traces, jobs can be cancelled intentionally. We group all these job outcomes as *aborted jobs* and collectively refer to failed and aborted jobs as ***unsuccessful jobs***.

There is another job outcome category. At LANL, users are required to specify a runtime estimate for each job. This estimate is treated as a time limit, similar to an SLO, and the scheduler kills the job if the limit is exceeded. We refer to these killings as ***timeout jobs*** and present them separately because they can produce useful work in three cases: (a) when HPC jobs use the time limit as a stopping criterion, (b) when job state is periodically checkpointed to disk, and (c) when a job completes its work before the time limit but fails to terminate cleanly.

Observation 8: *Unsuccessful job terminations in the Google trace are 1.4-6.8x higher than in other traces. Unsuccessful jobs at LANL use 34-80% less CPU time.*

In Figure 7, we break down the total number of jobs (left), as well as the total CPU time consumed by all jobs by job outcome (right). First, we observe that the fraction of unsuccessful jobs is significantly higher (1.4-6.8x) for the Google trace, than for the other traces. This comparison ignores jobs that timeout for Mustang, because as we explained above, it is unlikely they represent wasted resources. We also note that almost all unsuccessful jobs in the Google trace were aborted. According to the trace

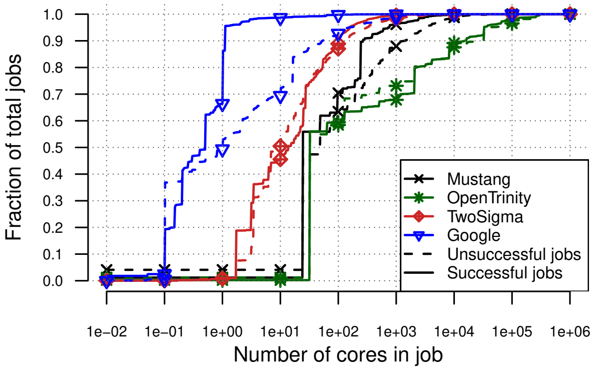


Figure 8: CDFs of job sizes (in CPU cores) for unsuccessful and successful jobs.

documentation [58] these jobs could have been aborted by a user or the scheduler, or by dependent jobs that failed. As a result, we cannot rule out the possibility that these jobs were linked to a failure. For this reason, prior work groups all unsuccessful jobs under the "*failed*" label [17], which we choose to avoid for clarity. Another fact that further highlights how blurred the line between failed and aborted jobs can be, is that all unsuccessful jobs in the TwoSigma trace are assigned a failure status. In short, our classification of jobs as "unsuccesful" may seem broad, but it is consistent with the liberal use of the term "failure" in the literature.

We also find that unsuccessful jobs are not equally detrimental to the overall efficiency of all clusters. While the rate of unsuccessful jobs for the TwoSigma trace is similar to the rate of unsuccessful jobs in the OpenTrinity trace, each unsuccessful job lasts longer. Specifically, unsuccessful jobs in the LANL traces waste 34-80% less CPU time than in the Google and TwoSigma traces. It is worth noting that 49-55% of CPU time at LANL is allocated to jobs that time out, which suggests that at least a small fraction of that time may become available through the use of better checkpoint strategies.

Observation 9: *For the Google trace, unsuccessful jobs tend to request more resources than successful ones. This is untrue for all other traces.*

In Figure 8, we show the CDFs of job sizes (in CPU cores) of individual jobs. For each trace, we show separate CDFs for unsuccessful and successful jobs. By separating jobs based on their outcome we observe that successful jobs in the Google trace request fewer resources, overall, than unsuccessful jobs. This observation has also been made in earlier work [17, 21], but it does not hold for our other traces. CPU requests for successful jobs in the TwoSigma and LANL traces are similar to requests made by unsuccessful jobs. This trend is opposite to what is seen in older HPC job logs [59], and since these traces were also collected through SLURM and MOAB

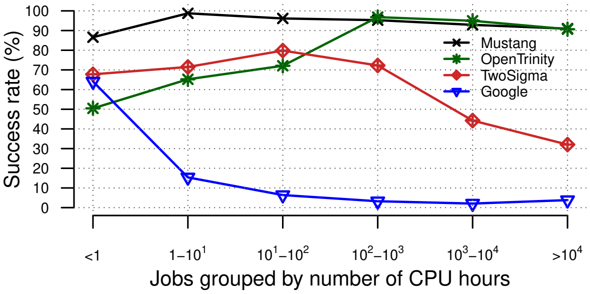


Figure 9: Success rates for jobs grouped by CPU hours.

we do not expect this discrepancy to be due to semantic differences in the way failure is defined across traces.

Observation 10: *For the Google and TwoSigma traces, success rates drop for jobs consuming more CPU hours. The opposite is true for LANL traces.*

For the traces we analyze, the root cause behind unsuccessful outcomes is not reliably recorded. Without this information, it is difficult to interpret and validate the results. For example, we expect that hardware failures are random events whose occurrence roughly approximates some frequency based on the components' Mean Time Between Failure ratings. As a result, jobs that are larger and/or longer, would be more likely to fail. In Figure 9 we have grouped jobs based on the CPU hours they consume (a measure of both size and length), and we show the success rate for each group. The trend that stands out is that success rates decrease for jobs consuming more CPU hours in the Google and TwoSigma traces, but they are increase and remain high for both LANL clusters. This could be attributed to larger, longer jobs at LANL being more carefully planned and tested, but it could also be due to semantic differences in the way success and failure are defined across traces.

Implications. The majority of papers analyzing the characteristics of job failures in the Google trace build failure prediction models that assume the existence of the trends we have shown on success rates and resource consumption of unsuccessful jobs. Chen et al. [9] highlight the difference in resource consumption between unsuccessful and successful jobs, and El-Sayed et al. [17] note that this is the second most influential predictor (next to early task failures) for their failure prediction models. As we have shown in Figure 9, unsuccessful jobs are not linked to resource consumption in other traces. Another predictor highlighted in both studies is job re-submissions, with successful jobs being resubmitted fewer times. We confirm that this trend is consistent across all traces, even though the majority of jobs (83-93%) are submitted exactly once. A final observation that does not hold true for LANL is that CPU time of unsuccessful jobs increases with job runtime [17, 22].

7 A case study on plurality and diversity

Evaluating systems against multiple traces enables researchers to identify practical sensitivities of new research and prove its generality. We demonstrate this through a case study on JVuPredict, the job runtime[2] predictor module of the JamaisVu [51] cluster scheduler. Our evaluation of JVuPredict with all the traces we have introduced revealed the predictive power of logical job names and consistent user behavior in workload traces. Conversely, we found it difficult to obtain accurate runtime predictions in systems that provide insufficient information to identify job re-runs. This section briefly describes the architecture of JVuPredict (Section 7.1) and our evaluation results (Section 7.2).

7.1 JVuPredict background

Recent schedulers [12, 24, 32, 51, 52] use information on job runtimes to make better scheduling decisions. Accurate knowledge of job runtimes allows a scheduler to pack jobs more aggressively in a cluster [12, 18, 54], or to delay a high-priority batch job to schedule a latency-sensitive job without exceeding the deadline of the batch job. In heterogeneous clusters, knowledge of a job's runtime can also be used to decide whether it is better to immediately start a job on hardware that is sub-optimal for it, let it wait until preferred hardware is available, or simply preempt other jobs to let it run [3, 52]. Such schedulers assume most of the provided runtime information is accurate. The accuracy of the provided runtime is important as these schedulers are only robust to a reasonable degree of error [52].

Traditional approaches for obtaining runtime knowledge are often as trivial as expecting the user to provide an estimate, an approach used in HPC environments such as LANL. As we have seen in Section 6, however, users often use these estimates as a stopping criterion (jobs get killed when they exceed them), specify a value that is too high, or simply fix them to a default value. Another option is to detect jobs with a known structure that are easy to profile as a means of ensuring accurate predictions, an approach followed by systems such as Dryad [30], Jockey [20], and ARIA [53]. For periodic jobs, simple history-based predictions can also work well [12, 32]. But these approaches are still inadequate for consolidated clusters without a known structure or history.

JVuPredict, the runtime prediction module of JamaisVu [51], aims to predict a job's runtime when it is submitted, using historical data on past job characteristics and runtimes. It differs from traditional approaches by attempting to detect jobs that repeat, even when successive runs are not declared as repeats. It is more effective, as only part of the history relevant to the newly submitted job is used to generate the estimate. To do this, it uses features of submitted jobs, such as user IDs and job names, to build multiple independent predictors. These predictors are then evaluated based on the accuracy achieved on historic data, and the most accurate one is selected for future predictions. Once a prediction is made, the new job is added to the history and the accuracy scores of each model are recalculated. Based on the updated scores a new predictor is selected and the process is repeated.

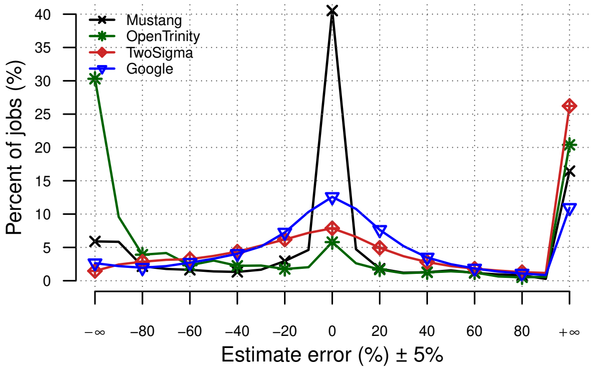


Figure 10: Accuracy of JVuPredict predictions of runtime estimates, for all four traces.

7.2 Evaluation results

JVuPredict had originally been evaluated using only the Google trace. Although predictions are not expected to be perfect, performance under the Google trace was reasonably good, with 86% of predictions falling within a factor of two of the actual runtime. This level of accuracy is sufficient for the JamaisVu scheduler, which further applies techniques to mitigate the effects of such mispredictions. In the end, the performance of JamaisVu with the Google trace is sufficient to closely match that of a hypothetical scheduler with perfect job runtime information and to outperform runtime-unaware scheduling [51]. This section repeats the evaluation of JVuPredict using our new TwoSigma and LANL traces. Our criterion for success is meeting or surpassing the prediction accuracy achieved with the Google trace.

A feature expected to effectively predict job repeats is the job's name. This field is typically anonymized by hashing the program's name and arguments, or simply by hashing the user-defined human-readable job name provided to the scheduler. For the Google trace, predictors using the logical job name field are selected most frequently by JVuPredict due to their high accuracy.

Figure 10 shows our evaluation results. On the x-axis we plot the prediction error for JVuPredict's runtime estimates, as a percentage of the actual runtime of the job. Each data point in the plot is a bucket representing val-

[2] The terms *runtime* and *duration* are used interchangeably here.

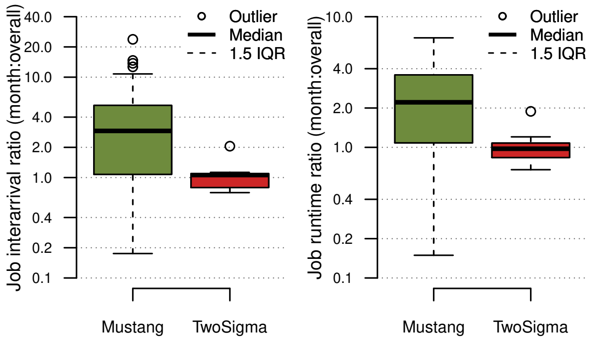


Figure 11: *Is a month representative of the overall workload?* The boxplots show distributions of the average job inter-arrival period (left) and duration (right) per month, normalized by the trace's overall average. Boxplot whiskers are defined at 1.5 times the distribution's Inter-Quartile Range (standard Tukey boxplots).

ues within 5% of the nearest decile. The y-axis shows the percentage of jobs whose predictions fall within each bucket. Overestimations of a job's runtime are easier to tolerate than underestimations, because they cause the scheduler to be more conservative when scheduling the job. Thus, the uptick at the right end of the graph is not alarming. For the Google trace, the total percentage of jobs whose runtimes are under-estimated is 32%, with 11.7% of underestimations being lower than half the actual runtime. We mark these numbers as acceptable, since performance of JVuPredict in the Google trace has been proven exceptional in simulation.

Although the logical job name is a feature that performs well for the Google trace, we find it is either unavailable, or unusable in our other traces. This is because of the difficulty inherent in producing an anonymized version of it, while maintaining enough information to distinguish job repeats. Instead, this field is either assigned a unique value for every job, or entirely omitted from the trace. All traces we introduce in this paper suffer from this limitation. The absence of the field, however, seems to not affect the performance of JVuPredict significantly. The fields selected by JVuPredict as the most effective predictors of job runtime for the Mustang and TwoSigma traces are: the ID of the user who submitted the job, the number of CPU cores requested by the job, or a combination of the two. We find that the TwoSigma workload achieves identical performance to Google: 31% of job runtimes are underestimated and 15% are predicted to be less than 50% of the actual runtime. The Mustang workload is much more predictable, though, with 38% of predictions falling within 5% of the actual runtime. Still, 16% of job runtimes were underestimated by more than half of the actual runtime. The similarity between the TwoSigma and Mustang results suggests that JamaisVu would also perform well under these workloads. Note that these results extend to the Google trace when the job name is omitted.

OpenTrinity performs worse than every other trace. Even though the preferred predictors are, again, the user ID and the number of CPU cores in the job, 55% of predictions have been underestimations. Even worse, 24% of predictions are underestimated by more than 95% of the actual runtime. A likely cause for this result is the variability present in the trace. We are unsure whether this variability is due to the short duration of the trace, or due to the workload being more inconsistent during the OpenScience configuration period.

In conclusion, two insights were obtained by evaluating JVuPredict with multiple traces. First, we find that although logical job names work well for the Google trace, they are hard to produce in anonymized form for other traces, so they may often be unavailable. Second, we find that in the absence of job names, there are other fields that can substitute for them and provide comparable accuracy all but the OpenTrinity trace. Specifically, the user ID and CPU core count for every job seem to perform best for both TwoSigma and the Mustang trace.

8 On the importance of trace length

Working with traces often forces researchers to make key assumptions as they interpret the data, in order to cope with missing information. A common (unwritten) assumption when using or analyzing a trace, is that it sufficiently represents the workload of the environment wherein it was collected. At the same time the Google trace spans only 29 days, while other traces we study in this paper are 3-60 times longer, even covering the entire lifetime of the cluster in the case of Mustang. Being unsure whether 29 days are sufficient to accurately describe a cluster's workload, we decided to examine how representative individual 29-day periods are of the overall workload in our TwoSigma and Mustang traces.

Our experiment consisted of dividing our traces in 29-day periods. For each such month we then compared the distributions of individual metrics against the overall distribution for the full trace. The metrics we considered were: job sizes, durations, and interarrival periods. Overall we found consecutive months' distributions to vary wildly for all these metrics. One distinguishable trend, however, is that during the third year the Mustang cluster is dominated by short jobs arriving in bursts.

Figure 11 summarizes our results by comparing the averages of different metrics for each month against the overall average across the entire trace. The boxplots show the distributions of average job interarrivals (left) and durations (right) per month, when normalized by the overall average for the trace. The boxplots are standard

Tukey boxplots, where the box is framed by the 25^{th} and 75^{th} percentiles, the dark line represents the median, and the whiskers are defined at 1.5 times the distribution's Inter-Quartile Range (IQR), or the furthest data point if no outliers exist (shown in circles here). We see that individual months vary significantly for the Mustang trace, and they differ somewhat less across months in the TwoSigma trace. More specifically, the average job interarrival of a given month can be 0.7-2.0x the value of the overall average in the TwoSigma trace, or 0.2-24x the value of the overall average in the Mustang trace. Average job durations can fluctuate between 0.7-1.9x of the average job duration in the TwoSigma trace, and 0.1-6.9x of the average in the Mustang trace. Overall, our results conclusively show that our cluster workloads display significant differences from month to month.

9 Related Work

The Parallel Workloads Archive (PWA) [19] hosts the largest collection of public HPC traces. At the time of this writing, 38 HPC traces have been collected between 1993 and 2015. Our HPC traces complement this collection. The Mustang trace is unique in a number of ways: it is almost two times longer in duration than the longest publicly available trace, contains four times as many jobs, and covers the entire lifetime of the cluster enabling longitudinal analyses. It is also similar in size to the largest clusters in PWA and its distribution of job duration distribution is shorter than all other HPC traces. The OpenTrinity trace is also complementary to existing traces, as it is collected on a machine almost two times bigger than the largest supercomputer with a publicly available trace (Argonne National Lab's Intrepid) as far as CPU core count is concerned.

Prior studies have looked at private cluster traces, specifically with the aim of characterizing MapReduce workloads. Ren et al. [42] examine three traces from academic Hadoop clusters in an attempt to identify popular application styles and characterize the input/output file sizes, the duration, and the frequency of individual MapReduce stages. These clusters handle significantly less traffic than the Google and TwoSigma clusters we examine. Interestingly, a sizable fraction of interarrival periods for individual jobs are longer than 100 seconds, which resembles our HPC workloads. At the same time, the majority of jobs last less than 8 minutes, which approximates the behavior in the Google trace. Chen et al. [10] look at both private clusters from Cloudera customers and Internet services clusters from Facebook. On the one hand, their private traces cover less than two months, while on the other hand their Facebook traces are much longer than the Google trace. Still, there are similarities in traffic, as measured in job submissions per hour. Specifically, Cloudera customers' private clusters deal with hundreds of job submissions per hour, a traffic pattern similar to the Two Sigma clusters, while Facebook handles upwards of a thousand submissions per hour, which is more related to traffic in the Google cluster. The diversity across these workloads further emphasizes the need for researchers to focus on evaluating new research using a diverse set of traces.

Other studies that look at private clusters focus on Virtual Machine workloads. Shen et al. [47] analyze datasets of monitoring data from individual VMs in two private clusters. They report high variability in resource consumption across VMs, but low overall cluster utilization. Cano et al. [5] examine telemetry data from 2000 clusters of Nutanix customers. The frequency of telemetry collection varies from minutes to days and includes storage, CPU measurements, and maintenance events. The authors report fewer hardware failures in these systems than previously reported in the literature. Cortez et al. [11] characterize the VM workload on Azure, Microsoft's cloud computing platform. They also report low cluster utilization and low variability in tenant job sizes.

Conclusion

We have introduced and analyzed job scheduler traces from two private and two HPC clusters. We publicly release all four traces, which we expect to be of interest to researchers due to their unique characteristics, including: the longest public trace to date spanning the entire 5-year lifetime of a cluster, one representing the largest cluster with a public trace to date, and the two longest private non-academic cluster traces made public to date.

Our analysis showed that the private clusters resemble the HPC workloads studied, rather than the popular Google trace workload, which is surprising. This observation holds across many aspects of the workload: job sizes and duration, resource allocation, user behavior variability, and unsuccessful job characteristics. We also listed prior work that relies too heavily on the Google trace's characteristics and may be affected.

Finally, we demonstrated the importance of dataset plurality and diversity in the evaluation of new research. For job runtime predictions, we show that using multiple traces allowed us to reliably rank data features by predictive power. We hope that by publishing our traces we will enable researchers to better understand the sensitivity of new research to different workload characteristics.

Dataset availability

The LANL Mustang, LANL OpenTrinity, and two Two Sigma scheduler logs can be downloaded from the ATLAS repository, which is publicly accessible through:

`www.pdl.cmu.edu/ATLAS`

Acknowledgments

We would like to thank John Wilkes, Steve Senator, our shepherd Konstantinos Karanasos, and the anonymous reviewers for their invaluable feedback. We also thank the members and companies of the PDL Consortium for their insights, feedback, and support: Alibaba Group, Broadcom, Dell EMC, Facebook, Google, HP Enterprise, Hitachi, IBM Research, Intel, Micron, Microsoft Research, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate Technology, Two Sigma, Toshiba, Veritas and Western Digital. A portion of this work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the U.S. Department of Energy contract AC52-06NA25396. The publication has been assigned the LANL identifier LA-UR-18-24488.

References

[1] ADAPTIVE COMPUTING. MOAB HPC Suite. `http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/`.

[2] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., AND STOICA, I. Effective Straggler Mitigation: Attack of the Clones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 185–198.

[3] BOUTIN, E., EKANAYAKE, J., LIN, W., SHI, B., ZHOU, J., QIAN, Z., WU, M., AND ZHOU, L. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 285–300.

[4] BURTSEV, A., RADHAKRISHNAN, P., HIBLER, M., AND LEPREAU, J. Transparent checkpoints of closed distributed systems in emulab. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), EuroSys '09, ACM, pp. 173–186.

[5] CANO, I., AIYAR, S., AND KRISHNAMURTHY, A. Characterizing Private Clouds: A Large-Scale Empirical Analysis of Enterprise Clusters. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 29–41.

[6] CARVALHO, M., CIRNE, W., BRASILEIRO, F., AND WILKES, J. Long-term SLOs for reclaimed cloud computing resources. In *ACM Symposium on Cloud Computing (SoCC)* (Seattle, WA, USA, 2014), pp. 20:1–20:13.

[7] CHEN, C., WANG, W., ZHANG, S., AND LI, B. Cluster Fair Queueing: Speeding up Data-Parallel Jobs with Delay Guarantees. In *Proceedings of the IEEE International Conference on Computer Communications* (May 2017), IEEE INFOCOM 2017.

[8] CHEN, S., GHORBANI, M., WANG, Y., BOGDAN, P., AND PEDRAM, M. Trace-Based Analysis and Prediction of Cloud Computing User Behavior Using the Fractal Modeling Technique. In *2014 IEEE International Congress on Big Data* (June 2014), pp. 733–739.

[9] CHEN, X., LU, C. D., AND PATTABIRAMAN, K. Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study. In *2014 IEEE 25th International Symposium on Software Reliability Engineering* (Nov 2014), pp. 167–177.

[10] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proc. VLDB Endow. 5*, 12 (Aug. 2012), 1802–1813.

[11] CORTEZ, E., BONDE, A., MUZIO, A., RUSSINOVICH, M., FONTOURA, M., AND BIANCHINI, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *ACM SIGOPS operating systems review* (2017), ACM.

[12] CURINO, C., DIFALLAH, D. E., DOUGLAS, C., KRISHNAN, S., RAMAKRISHNAN, R., AND RAO, S. Reservation-based Scheduling: If You're Late Don't Blame Us! In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 2:1–2:14.

[13] DELGADO, P., DIDONA, D., DINU, F., AND ZWAENEPOEL, W. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 497–509.

[14] DELGADO, P., DINU, F., KERMARREC, A.-M., AND ZWAENEPOEL, W. Hawk: Hybrid Datacenter Scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, 2015), USENIX Association, pp. 499–510.

[15] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 127–144.

[16] EIDE, E., STOLLER, L., AND LEPREAU, J. An Experimentation Workbench for Replayable Networking Research. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association, pp. 16–16.

[17] EL-SAYED, N., ZHU, H., AND SCHROEDER, B. Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (June 2017), pp. 1333–1344.

[18] FEITELSON, D. G., RUDOLPH, L., AND SCHWIEGELSHOHN, U. *Parallel Job Scheduling — A Status Report*. Springer Berlin Heidelberg, 2005, pp. 1–16.

[19] FEITELSON, D. G., TSAFRIR, D., AND KRAKOV, D. Experience with using the Parallel Workloads Archive. *Journal of Parallel and Distributed Computing 74*, 10 (2014), 2967 – 2982.

[20] FERGUSON, A. D., BODIK, P., KANDULA, S., BOUTIN, E., AND FONSECA, R. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 99–112.

[21] GARRAGHAN, P., MORENO, I. S., TOWNEND, P., AND XU, J. An Analysis of Failure-Related Energy Waste in a Large-Scale Cloud Environment. *IEEE Transactions on Emerging Topics in Computing 2*, 2 (June 2014), 166–180.

[22] Ghit, B., and Epema, D. Better Safe Than Sorry: Grappling with Failures of In-Memory Data Analytics Frameworks. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2017), HPDC '17, ACM, pp. 105–116.

[23] Gog, I., Schwarzkopf, M., Gleave, A., Watson, R. N. M., and Hand, S. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 99–115.

[24] Grandl, R., Chowdhury, M., Akella, A., and Ananthanarayanan, G. Altruistic scheduling in multi-resource clusters. In *OSDI* (2016), pp. 65–80.

[25] Gu, J., Lee, Y., Zhang, Y., Chowdhury, M., and Shin, K. G. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 649–667.

[26] Hennessy, J. L., and Patterson, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[27] Hibler, M., Ricci, R., Stoller, L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., and Lepreau, J. Large-scale virtualization in the emulab network testbed. In *USENIX 2008 Annual Technical Conference* (Berkeley, CA, USA, 2008), ATC'08, USENIX Association, pp. 113–128.

[28] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 295–308.

[29] Hung, C.-C., Golubchik, L., and Yu, M. Scheduling Jobs Across Geo-distributed Datacenters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 111–124.

[30] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 59–72.

[31] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the 22nd Symposium on Operating Systems Principles* (October 2009), SOSP '15, ACM, pp. 261–276.

[32] Jyothi, S. A., Curino, C., Menache, I., Narayanamurthy, S. M., Tumanov, A., Yaniv, J., Mavlyutov, R., Goiri, Í., Krishnan, S., Kulkarni, J., et al. Morpheus: towards automated slos for enterprise clusters. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation* (2016), USENIX Association, pp. 117–134.

[33] Leverich, J., and Kozyrakis, C. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 4:1–4:14.

[34] Liu, J., Shen, H., and Chen, L. CORP: Cooperative Opportunistic Resource Provisioning for Short-Lived Jobs in Cloud Systems. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)* (Sept 2016), pp. 90–99.

[35] Ma, J., Sui, X., Sun, N., Li, Y., Yu, Z., Huang, B., Xu, T., Yao, Z., Chen, Y., Wang, H., Zhang, L., and Bao, Y. Supporting Differentiated Services in Computers via Programmable Architecture for Resourcing-on-Demand (PARD). In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 131–143.

[36] Marshall, P., Keahey, K., and Freeman, T. Improving Utilization of Infrastructure Clouds. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (Washington, DC, USA, 2011), CCGRID '11, IEEE Computer Society, pp. 205–214.

[37] Massey Jr., F. J. The Kolmogorov-Smirnov Test for Goodness of Fit. *Journal of the American Statistical Association 46*, 253 (1951), 68–78.

[38] Meneses, E., Ni, X., Jones, T., and Maxwell, D. Analyzing the Interplay of Failures and Workload on a Leadership-Class Supercomputer. In *Cray User Group Conference (CUG)* (April 2015).

[39] Office of Science (DOE-SC) and the National Nuclear Security Administration (NNSA), U.S. Department of Energy. Exascale Computing Project. `https://exascaleproject.org/`.

[40] Rampersaud, S., and Grosu, D. Sharing-Aware Online Virtual Machine Packing in Heterogeneous Resource Clouds. *IEEE Transactions on Parallel and Distributed Systems 28*, 7 (July 2017), 2046–2059.

[41] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 7:1–7:13.

[42] Ren, K., Kwon, Y., Balazinska, M., and Howe, B. Hadoop's Adolescence: An Analysis of Hadoop Usage in Scientific Workloads. *Proc. VLDB Endow. 6*, 10 (Aug. 2013), 853–864.

[43] Ros, A., Chen, L. Y., and Binder, W. Predicting and Mitigating Jobs Failures in Big Data Clusters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (May 2015), pp. 221–230.

[44] Ros, A., Chen, L. Y., and Binder, W. Understanding the Dark Side of Big Data Clusters: An Analysis beyond Failures. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (June 2015), pp. 207–218.

[45] SchedMD. SLURM Workload Manager. `https://slurm.schedmd.com/`.

[46] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)* (Prague, Czech Republic, 2013), pp. 351–364.

[47] Shen, S., v. Beek, V., and Iosup, A. Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (May 2015), pp. 465–474.

[48] SNIR, M., WISNIEWSKI, R. W., ABRAHAM, J. A., ADVE, S. V., BAGCHI, S., BALAJI, P., BELAK, J., BOSE, P., CAPPELLO, F., CARLSON, B., CHIEN, A. A., COTEUS, P., DEBARDELEBEN, N. A., DINIZ, P. C., ENGELMANN, C., EREZ, M., FAZZARI, S., GEIST, A., GUPTA, R., JOHNSON, F., KRISHNAMOORTHY, S., LEYFFER, S., LIBERTY, D., MITRA, S., MUNSON, T., SCHREIBER, R., STEARLEY, J., AND HENSBERGEN, E. V. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications 28*, 2 (2014), 129–173.

[49] THE APACHE SOFTWARE FOUNDATION. Apache Spark. `https://spark.apache.org/`.

[50] THINAKARAN, P., GUNASEKARAN, J. R., SHARMA, B., KANDEMIR, M. T., AND DAS, C. R. Phoenix: A Constraint-Aware Scheduler for Heterogeneous Datacenters. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (June 2017), pp. 977–987.

[51] TUMANOV, A., JIANG, A., PARK, J. W., KOZUCH, M. A., AND GANGER, G. R. JamaisVu: Robust Scheduling with Auto-Estimated Job Runtimes. Tech. Rep. CMU-PDL-16-104, Carnegie Mellon University, September 2016.

[52] TUMANOV, A., ZHU, T., PARK, J. W., KOZUCH, M. A., HARCHOL-BALTER, M., AND GANGER, G. R. TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 35:1–35:16.

[53] VERMA, A., CHERKASOVA, L., AND CAMPBELL, R. H. ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing* (New York, NY, USA, 2011), ICAC '11, ACM, pp. 235–244.

[54] VERMA, A., KORUPOLU, M., AND WILKES, J. Evaluating job packing in warehouse-scale computing. In *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014* (11 2014), pp. 48–56.

[55] VERMA, A., PEDROSA, L., KORUPOLU, M. R., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France, 2015).

[56] WANG, W., LI, B., LIANG, B., AND LI, J. Multi-resource Fair Sharing for Datacenter Jobs with Placement Constraints. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2016), pp. 1003–1014.

[57] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), USENIX Association, pp. 255–270.

[58] WILKES, J. More Google cluster data. Google research blog, Nov. 2011. Posted at `http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html`.

[59] YUAN, Y., WU, Y., WANG, Q., YANG, G., AND ZHENG, W. Job failures in high performance computing systems: A large-scale empirical study. *Computers & Mathematics with Applications 63*, 2 (2012), 365 – 377.

SLAOrchestrator: Reducing the Cost of Performance SLAs for Cloud Data Analytics

Jennifer Ortiz[†], Brendan Lee[†], Magdalena Balazinska[†], Johannes Gehrke[*] and Joseph L. Hellerstein[‡]
[†] University of Washington Department of Computer Science & Engineering,[*]Microsoft,[‡]eScience Institute

Abstract

SLAOrchestrator is a new system designed to reduce the price increases necessary to support performance SLAs in cloud analytics systems. SLAOrchestrator is designed for SLAs that guarantee per-query execution times. Its core architecture consists of a double learning loop that improves both SLAs and resource management over time. It further utilizes an efficient combination of elastic query scheduling and multi-tenant resource provisioning algorithms to reduce the costs of performance guarantees.

1 Introduction

A variety of shared-nothing systems for data analytics are available as cloud services today, including Amazon Elastic MapReduce (EMR) [5], Amazon Redshift [4], Azure's HDInsight [8], and Azure Data Lake Analytics [46]. When using those systems, users upload their data to the cloud and issue queries on that data. Queries can include relational operators and various user-defined computations. A key challenge with these services, however, is that users must decide on a desired configuration: how many *service instances* they want to pay for and how powerful these instances should be.

The service configuration dramatically impacts price [2] and performance [53], yet it is known to be very difficult for users to select correctly [24]. Since users do not know what configuration to purchase, one approach is to offer performance-based service level agreements (SLAs), where the system promises to meet a given per-query latency or pay a penalty [41, 42].

Previous research has addressed the challenge of selecting and enforcing SLAs in various ways. One line of work assumes each tenant fits on a single server and the challenge is to pack tenants on a restricted set of servers [17, 34, 47], migrating tenants as needed [16], ordering queries for execution [12, 36], controlling admission [56, 42], and dispatching queries to servers [11, 37]. Other approaches assume the workload is known and require profile runs of queries, possibly restricted to processing samples [53, 25, 18, 23]. Knowledge of the workload and profile runs are reasonable assumptions in a transaction-processing system with a fixed set of stored procedures or in an analytics system that runs predefined reports, but not for ad-hoc analytical workloads.

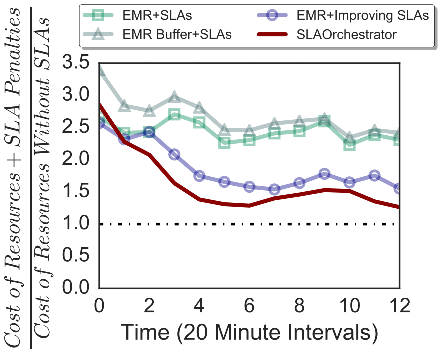


Figure 1: **A time-changing set of tenants executes ad-hoc, analytical queries subject to performance SLAs. Static resource allocation (EMR+SLAs), even with a buffer (EMR+SLA+Buffer) leads to large cost increases. Our improving SLAs (EMR+Improving SLAs), especially with multi-tenancy and other optimizations (SLAOrchestrator), bring costs down.**

Another line of work focuses purely on enforcing SLAs, assuming that SLAs are pre-defined [12, 11, 56]. SLA runtimes are artificially generated by, for example, offering a performance guarantee 10x the true latency [12], or by setting SLAs to be the performance of past executions [29]. Without the right SLAs, the best enforcement does not help: If the cloud provider overprovisions the underlying system, the user has to bear large costs, making the cloud provider less competitive and encouraging the user to take her business elsewhere. If the cloud provider underprovisions the underlying system, the cloud provider has to pay penalties for missed SLAs and thus loses money in the long term or must raise prices to compensate.

In this paper, we address the problem of *selecting* and *enforcing* SLAs for ad-hoc analytical queries over systems with multiple nodes. We develop SLAOrchestrator, a system that enables a cloud provider to offer query-level, performance SLAs for ad-hoc data analytics. Instead of relying on outside-generated SLAs [12, 11, 56], SLAOrchestrator uses our PSLAManager from prior work [41] to show the user what is possible and the price tag associated with various options. SLAOrchestrator generates, updates over time, and enforces SLAs in a way that successfully brings down the cost, close to that of the original service without SLAs.

Figure 1 shows our system in action given a set of random tenants and EC2 prices.[1] The x-axis shows time and the y-axis shows the ratio of the service cost with SLAs to the service cost without SLAs. When we add performance SLAs to Amazon EMR and let the cloud provision the number of Virtual Machines (VMs) purchased under the covers, costs grow dramatically either due to SLA violations (EMR+SLAs) or over-provisioning (EMR+SLAs+Buffer). Since guarantees depend on the quality of the SLAs (measured by how close runtime estimates are to the real runtimes on the purchased resources), a key component of our approach is *to improve SLAs over time* (EMR+Improving SLAs). We complement these improving SLAs with *novel resource scheduling and provisioning algorithms* that minimize costs due to over- or under-provisioning given a per-query SLA (SLAOrchestrator).

SLAOrchestrator achieves its goal through three key techniques that form the core contributions of this work. First, SLAOrchestrator is designed on the core idea of a double nested, learning loop. In the outer loop, every time a tenant arrives, the system generates a performance SLA given its current model of query execution times. That model improves over time as more tenants use the system. The SLA is in effect for the duration of a *query session*, which is the time from the moment a user purchases an SLA and issues their first query until the user stops their data analysis and leaves the system. In the inner loop, SLAOrchestrator continuously learns from user workloads to improve query scheduling and resource provisioning decisions and reduce costs during query sessions. To drive this inner loop, we introduce a new subsystem, that we call *PerfEnforce*. We present the overall system architecture in Section 2.

Second, the PerfEnforce subsystem comprises a new type of query scheduler. Unlike traditional schedulers, which must arbitrate resource access and manage contention, PerfEnforce's scheduler operates in the context of seemingly unbounded, elastic cloud resources. Its goal is *cost-effectiveness*. It schedules queries in a manner that minimizes over- and under-provisioning overheads. We develop and evaluate four variants of the scheduler. The first variant is based on a PI controller. Two variants model the problem as either a contextual or non-contextual multi-armed bandit (MAB) [50]. The last variant models the problem as an online learning problem. We present the query scheduler in Section 3.

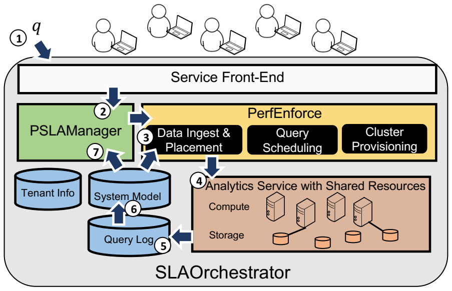


Figure 2: **SLAOrchestrator Architecture.**

Third, PerfEnforce also includes a new resource provisioning component. We evaluate two variants of resource provisioning: The first one strives to maintain a desired resource utilization level. The other one observes tenant query patterns and adjusts, accordingly, both the size of the overall resource pool and the tuning parameters of the query scheduler above. We present the resource provisioning algorithms in Section 4.

We evaluate all techniques in Section 5 and discuss related work in Section 6. As Figure 1 shows, SLAOrchestrator is able to reduce the costs associated with performance guarantees, bringing those costs down close to the basic service costs without guarantees.

2 System Architecture

Figure 2 shows SLAOrchestrator's system architecture. In this section, we present the details of that architecture and SLAOrchestrator's double nested learning loop.

2.1 System Components

SLAOrchestrator runs on top of a distributed, shared-nothing, data management and analytics engine (Analytics Service) such as Spark [7] or Hive [26]. We use our own Myria system [54] in the evaluation. Similar to how tenants use Amazon EMR today, in SLAOrchestrator, tenants upload their data to the service and analyze it by issuing declarative queries. While modern systems support complex queries, in this paper, we focus on relational select-project-join queries as proof-of-concept. However, there is nothing in our approach that precludes more complex queries in principal. On top of the Analytics Service, SLAOrchestrator includes an SLA generator (PSLAManager [41]), which generates performance SLAs for tenants. It also contains a dynamic scaling engine (PerfEnforce), which drives the scheduling and provisioning decisions for the underlying Analytics Service.

Analytics Service The back-end Analytics Service executes on a dynamically resizable pool of virtual

[1] We present the detailed experimental setup in Section 5 and the exact SLA function in Section 3.1.

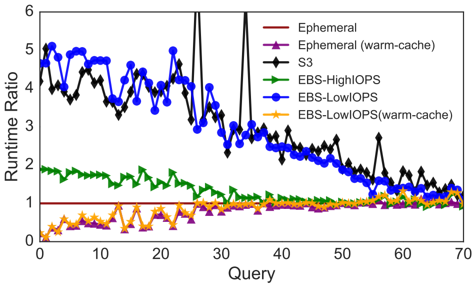
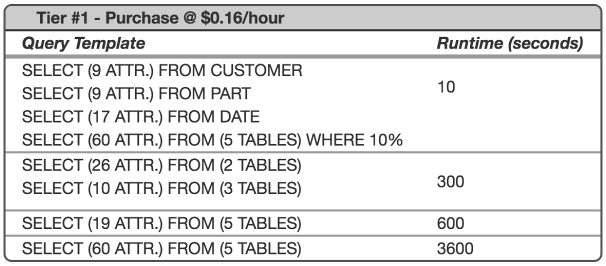


Figure 3: **Runtimes Compared to Local Storage.**



Tier #1 - Purchase @ $0.16/hour	
Query Template	*Runtime (seconds)*

SELECT (9 ATTR.) FROM CUSTOMER
SELECT (9 ATTR.) FROM PART
SELECT (17 ATTR.) FROM DATE
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 10%
	10

SELECT (26 ATTR.) FROM (2 TABLES)
SELECT (10 ATTR.) FROM (3 TABLES)
	300

SELECT (19 ATTR.) FROM (5 TABLES)	600
SELECT (60 ATTR.) FROM (5 TABLES)	3600

Figure 4: **Example performance SLA provided by PSLAManager with one service tier. Additional service tiers would show similar query templates but with different prices and performance thresholds.**

machines (VMs) running a resource manager such as YARN [6]. PerfEnforce uses that engine in a multi-tenant fashion and takes over all query scheduling decisions. When a tenant executes a query, PerfEnforce's query scheduling algorithm determines the number of containers needed to run the query. It then allocates that number of containers from the shared VM pool. Additionally, PerfEnforce's resource provisioning determines when to grow or shrink the pool.

Analytics Service: Tenant Isolation As is common in today's big data systems, each parallel partition of each query is a task that executes in a separate container. Each query submitted by each tenant thus gets allocated its own set of containers across the VMs. Furthermore, our design partitions each tenant's dataset and attaches individual data partitions to containers, allowing for a more isolated environment. In our experiments, we use YARN containers. We schedule one container per VM and thus use the terms interchangeably.

Analytics Service: Storage Once a user purchases an SLA and before they can query their data, PerfEnforce prepares their data by ingesting it into fast networked storage, EBS volumes in our prototype. Figure 3 motivates our choice. The figure shows the median query execution times across three runs for a variety of storage options available on Amazon Web Services (AWS). The y-axis shows the runtime relative to local storage. Queries on the x-axis are sorted by local storage runtimes in ascending order. The 70 queries shown are based on a 100SF TPC-H SSB dataset on Myria [54] running on 32 i2.xlarge instances. As the figure shows, fast networked storage, such as EBS-HighIOPS, provides performance competitive with ephemeral storage, even on a cold cache query, without the need to dynamically migrate (or replicate) data fragments as VMs are added and removed from the shared pool. This type of storage is also affordable at less than 20% of the cost of a VM. Because we seek dynamism and must support data-intensive processing, fast networked storage is appealing.

During query execution, PerfEnforce attaches EBS volumes to different VMs and detaches them as needed. Each EBS volume holds a partition of the data, resulting in a standard shared-nothing configuration. To avoid data shuffling overheads due to scaling, PerfEnforce ingests multiple copies of each table. Each copy is partitioned across a subset of EBS volumes such that, when a query executes over a set of k containers, it uses the version of its data spread across k EBS volumes. Due to space constraints, we refer to our technical report for further details on EBS data placement and its negligible impact on performance [43].

SLA Generation To generate SLAs, we use a system from our prior work, the PSLAManager [41], but our system could work with others. PSLAManager takes as input a database schema and statistics associated with a database instance for a tenant (we use the term user and tenant interchangeably). It generates a performance-based SLA specific to a database instance as shown in Figure 4 for the TPC-H Star Schema Benchmark [40]. Each tier has a fixed hourly price, which maps to a predefined set of storage and compute resources, along with sets of grouped queries where each group contains a time threshold ("Runtime" in the figure). The time threshold represents the performance guarantee for its respective group of queries and corresponds to *query time estimates* made by the SLA generator for the corresponding resource configuration. For each resource configuration, we only consider varying the number of instances, but consistently use a standard network, and EBS-HighIOPS for storage across all configurations.

Each tier represents a performance summary for a specific set of containers the service can use for tenant queries, which we call a *configuration*. Tiers can correspond to different types and numbers of containers, but we use a single type in our experiments. We refer to all possible configurations that the system can use to execute a query as the set $configs$. For example, $config = \{2, 4, \ldots, 64\}$, represents all even numbers of containers up to a maximum of 64. The system shows tiers for a pre-defined subset of these configurations. Later, it can schedule queries using the full set of configurations. The price of each tier is at least the sum of the hourly cost of the containers and network storage.

When a tenant purchases a performance SLA, she unknowingly purchases a configuration. The system starts a *query session* for the tenant and the latter starts paying the corresponding fixed hourly price. During the session, the tenant issues queries. The queries get queued up and execute one after the other, each one running in the entire set of containers in the purchased configuration. As we present in Section 3, PerfEnforce changes these allocations over time based on how fast they execute compared with the initial SLA time.

2.2 Double Nested Learning

To drive the SLA generation, SLAOrchestrator maintains a log of all past queries executed in the system. Initially, it executes queries from a 100GB dataset generated by the Parallel Data Generation Framework(PDGF) [45]. The system runs queries on all configurations that it will sell to populate the query log. With this information, SLAOrchestrator builds a model of query execution times. Each query is represented by a feature vector. Features correspond to query plan properties including the number of tables being joined, their sizes, the query cost estimates from the query optimizer, the number of containers in the configuration, etc. SLAOrchestrator learns a function from that feature vector to a query execution time. In our work, we use a simple linear model as in prior work [41, 53]. More complex models are possible but we find a simple linear model to yield good results for the select-project-join queries that we focus on in this paper. With this model, predictions are made by learning the coefficients (a weight vector, w) [9] given the query features, x_q: $y(x_q, w) = \sum_{d=1}^{D} w_d \cdot x_{q_d}$.

With our previous PSLAManager work [41], we observed that when a new tenant joins the system, estimates for that tenant's queries are likely to be inaccurate because the system has limited information about the tenant data and queries (only statistics on base data). However, as the tenant starts to execute queries, the system can quickly learn the properties of the data and can specialize its model to that data. PerfEnforce uses this information to dynamically adjust query scheduling and resource provisioning decisions in the context of an existing SLA. We call this the **Inner Learning Loop**. The effect of this learning is also that the system updates the SLA that it offers after each query session. This is SLAOrchestrator's **Outer Learning Loop**. The benefit of more precise SLAs to tenants is the overall reduction in the service cost. We use TensorFlow [1] to build this model and train on the PDGF dataset. We generate 4000 queries (500 per configuration) and record the features as well as runtimes into the System Model.

Figure 2 shows in more detail how SLAOrchestrator components interact with one another. Steps 1 through 6 denote the **Inner Learning Loop**: (1) Each tenant query, q is issued through the service front-end. (2) PSLAManager determines q's promised SLA time based on the service tier that the user previously purchased. (3) PerfEnforce uses query scheduling algorithms in conjunction with the System Model to determine the number of containers to schedule for q. (4) PerfEnforce schedules q on the Analytics Service. (5) The Analytics Service sends metadata about the query to the Query Log. (6) The System Model parses the Query Log metadata and stores features for the learning models. Once a tenant completes their session, SLAOrchestrator initiates the **Outer Learning Loop**. In Step 7, the PSLAManager system takes the information from the System Model and generates an improved SLA.

In the next two sections, we focus on the PerfEnforce subsystem and its query scheduling (Section 3) and resource provisioning (Section 4) algorithms, which are part of SLAOrchestrator's inner learning loop.

3 Dynamic Query Scheduling

Every time a new tenant purchases a service tier, PerfEnforce begins a query session for that tenant. The initial state of the query session indicates the configuration (i.e., number of containers) that corresponds to the purchased service tier. Many sessions are active at the same time and PerfEnforce receives streams of queries from these active tenants. Each query is associated with a possibly imperfect SLA. That is, the query may run significantly faster or slower than the SLA time if scheduled on the purchased set of containers. PerfEnforce's goal is to determine how many containers to actually use for each query with the goal to minimize operation costs. In this section, we present PerfEnforce's query scheduling algorithm.

3.1 Optimization Function

Consider a cloud service operation interval $T = [t_{start}, t_{end}]$. The total operating cost to the cloud during that interval is the cost of the resources used for the service and the cost associated with SLA violations for tenants active during that interval. Thus, PerfEnforce's goal is to minimize the following cost function :

$$\text{cost}(T) = \text{cost}_R(T) + \sum_{u \in U(T)} (\text{penalty}(u)) \qquad (1)$$

where $U(T)$ is the set of all tenants active during time interval, T, and $cost_R(T)$, is given by:

$$\text{cost}_R(T) = \sum_{t=t_{start}}^{t_{end}-1} \text{cost}_t(\text{resources}) \qquad (2)$$

where $cost_t(resources)$ represents the cost of resources for time interval $[t, t+1]$, which depends on the size and the price of individual compute instances.

The SLA penalty, $penalty(u)$, is the amount of money to refund to user u in case there are any SLA violations. In this paper, we use the following formulation:

$$\mathcal{S}\left(\frac{1}{|\mathcal{W}_u|} \sum_{q \in \mathcal{W}_u} \max(0, \frac{t_{real}(q) - t_{sla}(q)}{t_{sla}(q)})\right) * \alpha * p_u \qquad (3)$$

where $\mathcal{W}_u$ is the sequence of queries executed by user u, $t_{real}(q)$ is the real query execution time of query q,

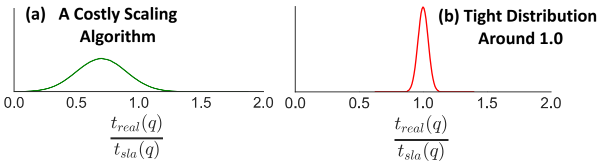


Figure 5: **Examples of Distributions of Query Performance Ratios**

$t_{sla}(q)$ is the SLA time of q, p_u is the session price paid by user u in the absence of SLA violations, and α is a configurable parameter that we vary in our experiments to adjust the cost of SLA penalties compared with container resource costs. $\mathcal{S}$ is a step function that rounds up and truncates values. This step function is inspired by real SLAs in cloud services that incur penalties based on availability outages [8, 49, 51].

3.2 Query Scheduling Algorithms

For each query $q \in \mathcal{W}_u$ and for each user u, PerfEnforce's query scheduling algorithm must determine the number of containers from the shared pool to allocate to the query. PerfEnforce begins with using the number of containers that corresponds to the purchased service tier. It observes the resulting query runtimes and dynamically adjusts the number of containers for subsequent queries by using a *scaling* algorithm. It runs a scaling algorithm separately for each tenant.

To minimize resource costs, the scaling algorithm should schedule queries on the smallest possible number of containers. To avoid SLA penalties, however, it must schedule queries on sufficiently large numbers of containers to ensure that the real query execution time, $t_{real}(q)$, is below the SLA time, $t_{sla}(q)$. We define the *query performance ratio* as $\frac{t_{real}(q)}{t_{sla}(q)}$ and the goal of the query scheduling algorithm is thus to execute each query in the configuration that yields a performance ratio of 1.0. In practice, if the query scheduling algorithm aims for query performance ratios of X, it will yield a query performance ratio distribution around X as illustrated in Figure 5. To illustrate our point, we plot synthetic Gaussians. Real distributions are noisier. Since we can adjust the mean of the distribution (a.k.a. setpoint), X, the quality of the scheduling algorithm is determined by the tightness of the distribution around X. In other words, if the distribution is wide (large standard deviation σ), then the system is either wasting resources for many queries (Figure 5a) or causing a large number of SLA violations. A good query scheduling algorithm should yield a tight distribution as in Figure 5b).

3.2.1 Reactive Scaling Algorithms

A reactive algorithm observes errors between the real and SLA runtimes and adjusts the number of containers accordingly for each subsequent query. We implement a Proportional Integral (PI) controller and a Multi-Armed-Bandit (MAB) as our reactive methods. Both of these techniques have successfully been used in other resource allocation contexts [31, 33, 35, 37].

A limitation of the these techniques is that the configuration size chosen for a new query depends only on the rewards or errors of previous queries ignoring the features of the current query. We use the reactive methods as baseline.

Proportional Integral Control (PI). Feedback control [28] in general, and PI controllers in particular, are commonly used to regulate a system in order to ensure that it operates at a given reference point. With a PI controller, at each time step, t, the controller produces an actuator value $u(t)$. In our scenario, this is the number of containers to use for the current query. The actuator value, causes the system to produce an output $y(t+1)$ at the next time step. We compute $y(t)$ as the average query performance ratio over some time window of queries w: $y(t) = \frac{1}{|w|}\sum_{q\in w}\frac{t_{real}(q_j)}{t_{sla}(q_j)}$ where $|w|$ is the number of queries in w. The goal is for the output, $y(t)$, to be equal to some desired reference output $r(t)$, 1.0 in our setting.

The error $e(t) = y(t) - r(t)$ captures a percent error between the current and desired average runtime ratios. Since the number of containers to spin up and remove given such a percent error depends on the configuration size, we add that size to the error computation as follows: $e(t) = (y(t) - r(t))u(t)$.

The PI controller, chooses the next number of containers as a combination of the initial configuration size $u(0)$, the most recently observed error, $e(t)$, and the sum of all accumulated errors $\sum_{x=0}^{t} e(x)$. k_p and k_i are tunable controller parameters, which determine how strongly the controller reacts to recent errors and how much it weighs history: $u(t+1) = u(0) + \sum_{x=0}^{t} k_i e(x) + k_p e(t)$

Multi-Armed Bandits (MAB). In a MAB problem, the system must repeatedly choose among k different options, or *arms*. At each timestep t, the system makes a decision by selecting one of k arms, a_t, and receives a reward, r_t [50]. In our setting, each arm is a configuration from the set $configs$. The arm choice is the decision to schedule the next query using a given configuration size.

The goal is to maximize the total reward across many timesteps. In the bandit setting, the algorithm must learn the reward distributions for different arms through a process of trial and error [9]. At each timestep, the system must thus choose to either select the arm with the highest estimated reward (*exploitation*) or try another arm (*exploration*) in order to acquire more information and maximizing the reward across all timesteps [50].

To help balance between exploration and exploitation, we use a heuristic known as *Thompson Sampling* [10]. During initialization, we define priors describing the expected reward of each arm. In our setting, we do not make assumptions for each configuration. Instead, we initialize the model for each arm using a uniform distribution, a noninformative prior. At timestep t, the system constructs a posterior distribution for each arm based on observed rewards, $P(\theta|a, r_0, ..., r_{t-1})$, where θ represents the model parameters. For each query submitted, the system samples from a candidate posterior distribution, defined as $\hat{\theta}$. Given that our prior is based on a uniform distribution, we use a t-distribution to represent our posterior. This t-distribution takes the reward mean, variance, and count as input. As the system samples from this posterior, we select the arm with the highest expected reward, $arg\,max_a \mathbb{E}[P(r_t|\hat{\theta}_\alpha)]$.

3.2.2 Proactive Scaling Algorithms

To address the limitations of the reactive techniques, we consider two other scaling algorithms that both include additional context, x_q, for each incoming query, where x_q is a $D-$dimensional vector of features describing the query, $x_q = (x_{q_1}, ..., x_{q_D})^T$. To generate the feature vector, we use the query optimizer of the back-end query execution engine and include information from the query plans (e.g. number of columns, estimated costs, estimated rows, estimated width, and the number of workers scheduled to run the query).

Contextual Multi-Arm Bandit (CMAB). This approach is a variant of the multi-armed bandit problem that includes contextual information. In a CMAB problem, at each timestep t, the algorithm receives a feature vector, x_q, as input, and uses it to determine the best arm, a_t. CMAB does this by building a model *for each configuration* that predicts the reward in that configuration given a query feature vector. The expected value of the reward for each arm and feature vector thus becomes: $q_\star(a) = \mathbb{E}[r_t|a_t, x_q, \theta]$.

Where θ represents the parameters of the generated model [10]. As with MAB, PerfEnforce uses the Thomson sampling heuristic to balance exploration and exploitation. At each timestep t, PerfEnforce builds a predictive model for each state by computing a bootstrap sample over all previous observations. PerfEnforce selects the action that corresponds to the state with the best predicted reward (i.e., reward closest to 1.0). In our prototype implementation, we use the REPTree model from Weka [22] as used in BanditDB [37]. For the first N queries in a tenant's session, we begin with a "warm-up" phase where we execute queries a small number of times in each configuration to initialize the observations for that configuration. PerfEnforce runs the "warm-up" session at the start of the query session, which could impact performance for some queries.

Online Learning The CMAB technique described above presents two practical challenges. First, it is difficult to determine the number of queries that should be used to initialize the distributions for each state. At least one query must be executed in each state, which can be either unnecessarily expensive or undesirably slow. The overhead especially penalizes short query sessions as early queries undergo larger amounts of exploration. Second, the observations collected are independent for each state. If one configuration suddenly results in slower or faster runtimes, this knowledge does not propagate to other states.

Because of the above limitations, we propose a different algorithm for our setting. We build a single model of query execution times with the configuration size as a feature. As a user executes queries, we always schedule those queries in configuration sizes expected to yield the best performance ratio and use the resulting query execution times to update our global model.

As described in the previous section, SLAOrchestrator maintains a model of query execution time that it uses for SLA generation. The idea here is for PerfEnforce to *continuously update* that model, during a tenant's query session, based on the measured query execution times. To update the model, PerfEnforce uses stochastic gradient descent. For each data point, it slowly updates the weight vector based on the gradient of a loss function, E: $w^{(\tau+1)} = w^{(\tau)} - \eta\nabla E$ [9]. Where τ represents the nth data point and η represents the learning rate. Importantly, PerfEnforce maintains a separate model of query execution time for each dataset so as to specialize its model to the properties of that dataset. If the underlying data significantly changes, the model could take time to adjust to changes, depending on the learning rate. Since we primarily focus on analytic sessions, we do not evaluate how this model adapts to updates. Training this model is relatively cheap, taking approximately 2.38s for a single epoch. Each prediction takes ∼10ms.

Setpoint Adjustment With all algorithms above, PerfEnforce strives to schedule queries such that their performance ratios form a tight distribution around a desired setpoint. An important question is how to tune the value of that setpoint. If the setpoint is 1.0 and the mean of the distribution falls on that setpoint, 50% of all queries will miss their SLA times. The setpoint can be lowered such that more, perhaps 90% of all queries, meet their SLA time. Lowering the setpoint, however, will increase the number of containers used for those queries and will thus raise resource costs. In SLAOrchestrator, we adjust the setpoint dynamically. We do so at the same time as we make cluster provisioning decisions as described next.

4 Dynamic Provisioning

With the above query scheduler, the total number of containers needed to service the active set of tenants varies over time. To reduce operation costs, PerfEnforce includes a resource provisioning component that determines when to grow and shrink the *shared* pool of compute resources. Provisioning is particularly challenging as it can take time to spin up new virtual machines. We observe that it takes approximately 12 seconds to spin up a virtual machine with a pre-loaded image on Amazon. We consider this start up time throughout our evaluation.

Utilization Provisioning: The most common approach to resource provisioning is to maintain a desired resource utilization level. Typically, this means adding (or removing) resources when CPU, I/O, network and memory usage move beyond (or below) set thresholds [3, 8, 21, 48].

We posit, however, that measuring resource utilization levels directly is not the right approach for PerfEnforce because tenants are allocated resource containers. As such, some tenants might execute I/O intensive workloads while others may run CPU intensive workloads, leading to very different resource utilization levels for various containers. In general, high resource utilization does not imply a higher demand for resources [14].

Instead of aiming for a given CPU or I/O utilization goal, we aim for an average *VM utilization* target, Z. The utilization of each machine is measured by the percentage of time it is actively running queries (wall clock time). For our target Z, we aim for an average utilization across all shared VMs, $AvgUtilization$.To determine the number of machines the system should provision to meet Z, we implement a PI controller where the set point is Z. Besides wall clock time, we note that other metrics for system state could be used as well. For example, the system could target a desired percentage of idle machines or a desired tenant query queue length.

Simulation-based Provisioning: For a more proactive approach to provisioning, we propose to explicitly consider tenant recent workloads rather than only measure resource utilization. Specifically, we propose to build models of tenant workloads and estimate the smallest number of shared resources to support these modeled workloads. This approach should be more effective than simply looking at utilization, since the latter is tightly coupled with the specific set of executed queries and the query scheduler's resource allocation decisions, which are themselves constrained by the amount of shared resources. To estimate the best number of shared resources to support tenant modeled workloads, we use simulations. This approach is not new and has been recently used in the "What-If" engine from Tempo [52], where the goal is to simulate the performance of many configurations of the MapReduce Resource Manager. We aim to understand how such a provisioning algorithm in combination with a learning query scheduler can help make profitable decisions in a multi-tenant service.

In this provisioning approach, we model each tenant, u, with a tuple (Q_u, λ_u), where Q_u is a set of queries that the tenant may issue and λ_u is the tenant's average think time between consecutive queries. PerfEnforce learns both values from a recent window of each tenant's query session. Based on these models, PerfEnforce then generates random sessions for each active tenant. To generate a random session, PerfEnforce samples queries from the recent query workload and also samples the think time based on a Poisson distribution. During these simulations, we also evaluate the costs of dynamically shifting the setpoint. In general, these simulations help PerfEnforce discover whether setpoint adjustments are necessary for active tenants or whether nodes should be added or removed to further save on costs.

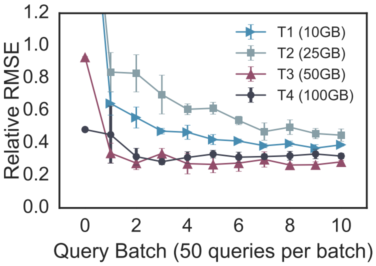


Figure 6: **SLA improvements across query sessions**

5 Evaluation

We run SLAOrchestrator and execute all queries on Amazon EC2 using i2.xlarge (4 ECU, 30 GB Memory) instance types priced at $0.12/hr. We consider eight types of query scheduling configurations, each with a different number of compute instances: `configs` $= \{4, 8, 12, 16, 20, 24, 28, 32\}$. For multi-tenant experiments, we run simulations with up to thousands of servers and use the query times measured on EC2. For our underlying shared-nothing, database management system, we use Myria [54]. Myria uses PostgreSQL as its storage subsystem.

To generate each tenant's query sequence, $\mathcal{W}_u$, we alternate between different patterns of queries. For example, one tenant might run small, lightweight queries for a majority of the session before switching to queries with larger joins and higher latencies. Thus, for some random number of queries, k, we define the following three discrete distributions: (1) number of joins, (2) number of projected attributes, and (3) selectivity factor. For the next k queries, we sample from each of these distributions and generate a query that meets all the sampled characteristics. Once k queries are generated, we define new distributions for the next random interval of queries. We use both uniform and skewed (zipfian) distributions. Unless stated otherwise, all the query workloads throughout the evaluation are generated in this fashion.

5.1 Evaluation of SLA Predictions

A key tenet of SLAOrchestrator is the idea that the system should update SLAs because they rapidly improve as a tenant queries a database. We validate this hypothesis in this section. Figure 6 shows the error of the SLA predictions for four tenants, each with a different, random star schema [45] and database instance: T1 = 10GB, T2 = 25GB, T3 = 50GB, T4= 100GB. We generate a set of SPJ queries with random selection predicates for each tenant. As tenants execute queries, SLAOrchestrator updates the query time model separately for each database. After each query batch, PSLAManager re-generates an updated SLA. As the figure shows, in all cases, the RRMSEs (relative root mean squared errors) between the real runtimes and the predicted SLA runtimes decreases rapidly after the first batch and then improves more slowly. We compute the error on a sample of queries generated by the PSLAManager for the tenant SLA. We measure the RRMSE as: $\sqrt{\frac{1}{|\mathcal{W}|}\sum_{q\in\mathcal{W}}(\frac{(t_{real}(q)-t_{sla}(q))}{t_{sla}(q)})^2}$. The prediction errors observed before running an initial batch of queries (Query Batch 0), are highly dependent on the similarity between the tenant's database and the synthetic database used to train our offline base model. In our experiments, while databases differ in their schemas and table sizes, we find that table sizes have the greatest impact on prediction errors. Our offline model is trained on a generated 100GB PDGF dataset. We observe a higher initial RRMSE error (approx. 2.4-2.5) for tenants T1 and T2 with the smaller databases.

5.2 Evaluation of Query Schedulers

The goal of each query scheduler is to ensure a tight distribution (small σ) of query performance ratios around a μ close to 1.0 (later in Section 5.4, we consider dynamic setpoint tuning). In this section, we evaluate how the different scheduling algorithms perform in the face of different tenant workloads. All tenant workloads are based on the 100GB TPC-H SSB benchmark [40]. We evaluate the algorithms using different-quality SLAs as shown in Table 1, which could correspond, for example, to different model qualities as shown in Section 5.1.

We first evaluate the PI-Control scheduling algorithm on four different SLA types and, in each case, on 10 different, randomly generated, tenant query sessions. We execute the PI controller on each tenant's query session independently and measure the resulting query performance distribution for that tenant. We then compute the average μ and σ across these 10 distributions and plot them in the first row of Figure 7. The y-axis represents the distance between μ and 1.0, while the x-axis displays the standard deviation of the query performance distributions. Because the PI controller has three tunable parameters (k_p, k_i and w), each point in the figure corresponds to one such parameter combination. For each graph, we also plot the average distribution of an Oracle, which always selects the best configuration size for each query. The best parameter combinations are those closest to the Oracle. If any technique's parameters result in a distribution with a higher σ or a μ farther from 1.0, this error impacts cost, which ultimately depends on the cost function. As the figure shows, for all SLAs, the PI-Control algorithm results in average distributions that are far from the average distribution of the Oracle. There are no best set of parameters that work across all workloads.

In the second row of Figure 7, we show the average distributions for MAB, CMAB, and online learning across the same set of SLA types and tenant query sessions. Note, the ranges for the axes are much smaller for these graphs compared to the PI-Controller, which shows that these techniques result in average distributions much closer to the Oracle. For both bandit techniques, we execute each tenant's query session 20 times due to their variance when sampling. For online learning, we vary the learning parameter, η. In the first 4 columns of the figure, we omit the performance ratios for the first 20 queries for all scaling techniques, since the bandits require an initial "warm-up" phase, where they need to try each configuration at least two times.

For the SG SLA, the bandit techniques result in average distributions nearly identical to the Oracle. Since both techniques rely on learning a distribution of query performance ratios per configuration, they quickly find the optimal configuration during the warm-up phase and select this configuration for a majority of the queries. Since the online learning technique is directly predicting the runtimes for each query, the prediction errors result in average distributions that are slightly farther away from the Oracle. For the PLJ SLA, all techniques perform similarly as most of the runtimes are meet at configurations that are close to the purchased tier. In contrast, online learning outperforms both bandit-based methods for the NLJ and Initial SLAs. The NLJ SLA underestimates runtimes, which requires the schedulers to accurately choose across a wider set of configuration options. For these more difficult cases, context is critical as the scheduling algorithm must make different decisions for different queries. There is no single best configuration. As a result, CMAB and the online learning approach both outperform the simpler MAB scheduler. Online learning further outperforms CMAB because this technique is able to quickly learn the performance correlations between configurations, which is crucial for the initial SLA as it requires scaling for each query.

The final column shows the average performance ratio distributions when using the initial SLA and including the queries in the warm-up period. As the figure shows, the online learning technique significantly outperform the bandit-based methods because it has the extra benefit of starting to learn from the offline model and learning more quickly because it learns a single model for all configurations. These results show that the PI controller is ill-suited to our problem and we do not consider

SLA	Description
Small Gaussian Error (SG SLA)	SLA assumes a good prediction model and tests sensitivity to small errors (or variance) in query times. Generated by taking the real query execution times at the purchased tier and adding a small Gaussian error: $\mathcal{N} = (\sigma = 0.1 * t_{real}(q), \mu = 0)$.
Positive/Negative Gaussian Errors for Large Joins (PLJ/NLJ SLA)	We skew SLA runtimes for some query types. We introduce large positive/negative errors to the real runtimes on queries with a large number of joins (>3 joins) and with a runtime $>$100 seconds. We update the runtime to $t_{real}(q) + \|e\|$ (or $-\|e\|$), where e is sampled from a Gaussian distribution, $\mathcal{N} = (\sigma = 0.3 * t_{real}(q), \mu = 0)$. For other queries, we still inject small errors as in SG SLA.
Initial SLA	This is the least accurate SLA, where runtimes are generated by an initial offline-trained model.

Table 1: **SLAs used in experiments.**

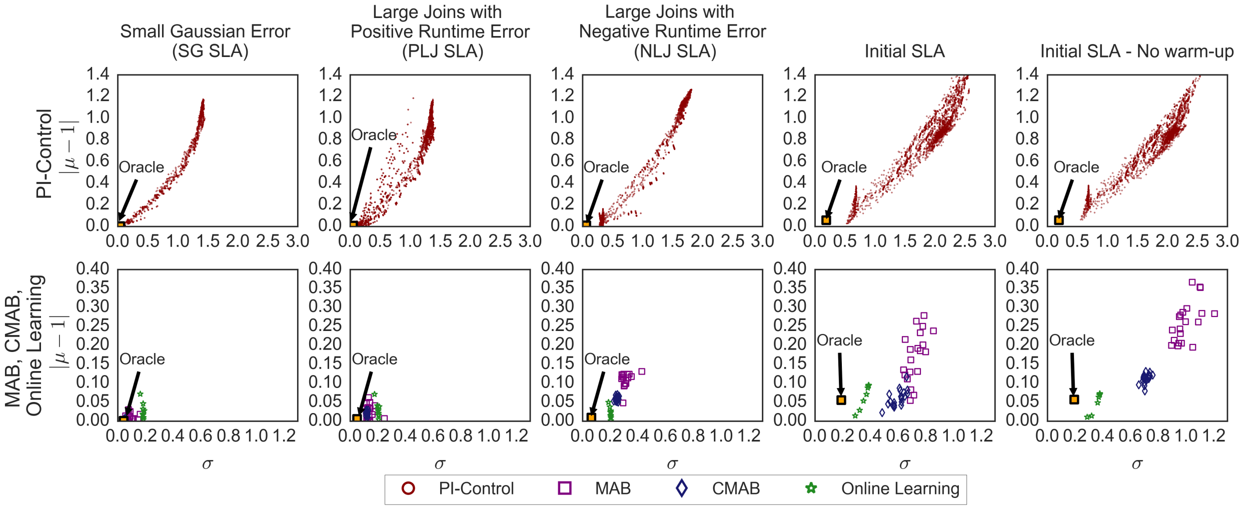


Figure 7: **Evaluation of PI-Controller, MAB, CMAB and online learning scheduling algorithms**

	MAB	CMAB	Online Learning	Oracle
μ	1.1368	1.1244	1.0161	1.0015
σ	0.1680	0.0871	0.0522	0.0008

Table 2: **Ratio distributions during slow down**

it further.

We now evaluate how query scheduling algorithms can adapt to changing conditions. Recall, the goal of these query schedulers is to ensure a query performance ratio distribution close to 1.0. We generate a query sequence by selecting one query and running it repeatedly several times. Each time we run this query, we record the query performance ratio. Once we reach the 250th iteration, we increase the query's runtime by 25% (essentially slowing down the system) for the rest of the session, running up to 1000 iterations. Table 2 shows that the online learning technique reacts the fastest to this change in conditions, leading to an overall mean performance ratio closest to 1.0. We omit additional experiments with different changing workloads due to space constraints.

5.3 Evaluation of Provisioning Algorithms

We first evaluate each provisioning algorithm in combination with the Oracle query scheduler to ensure that query runtime penalties are not a side-effect of the query scheduler's mispredictions. We launch each multi-tenant tenant session based on session parameters summarized in Table 3.

We introduce up to 100 tenants in a session and simulate a shared cluster with thousands of containers/VMs. We sample arrival times, think times, and session durations from Poisson distributions defined by their corresponding parameters $\lambda_{arrival}$, $\lambda_{thinktime}$ and $\lambda_{terminate}$. PerfEnforce always keeps at least a minimum of 4 machines launched at all times, to ensure that there are enough machines available to execute queries. Each provisioning algorithm monitors the shared resources and tenants for $\mathcal{M}$ minutes before adding or removing VMs from the pool. Our step function $\mathcal{S}$ provides no service credit if the system misses the runtime by 10%. For each additional 20% increment and given a threshold from x% to y%, we increase the credit to y%.

Notation	Description
U_{init}	Initial number of tenants in the session
V_{init}	Initial number of virtual machines
$\lambda_{arrival}$	Average time between new tenants
$\lambda_{thinktime}$	Average tenant think time
$\lambda_{terminate}$	Average tenant session duration
$\mathcal{M}$	Provisioning monitoring time interval

Table 3: **Parameters of multi-tenant experiments**

As described in Section 2, each submitted query gets allocated a set of containers. We schedule one container (running a Myria process) per VM. For each query, the system assigns the tenant's EBS volumes to a set of VMs in the pool. After the query completes, the volumes are detached from the VMs, making them available to other tenants. We find it takes 4 seconds to mount a volume to

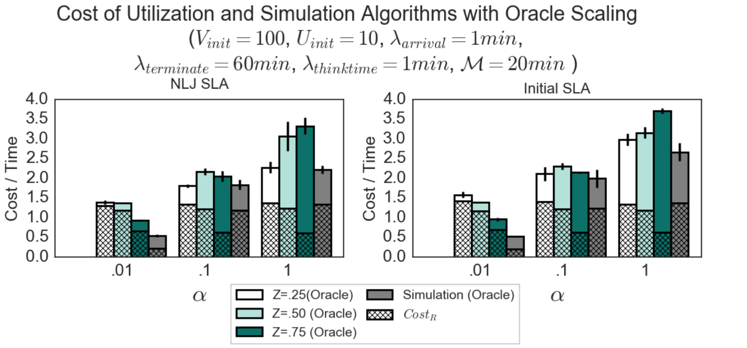


Figure 8: **Comparing utilization-based and simulation-based provisioning in conjunction with an Oracle query scheduler. The hash pattern represents the proportion of the cost due to $Cost_R$**

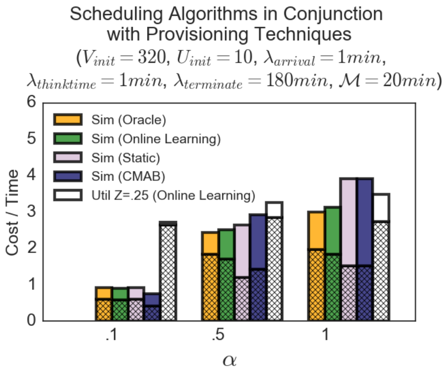


Figure 9: **Costs of query scheduling in conjunction with provisioning**

a VM. Detaching takes approximately 11 seconds. We include these delays in the experiments.

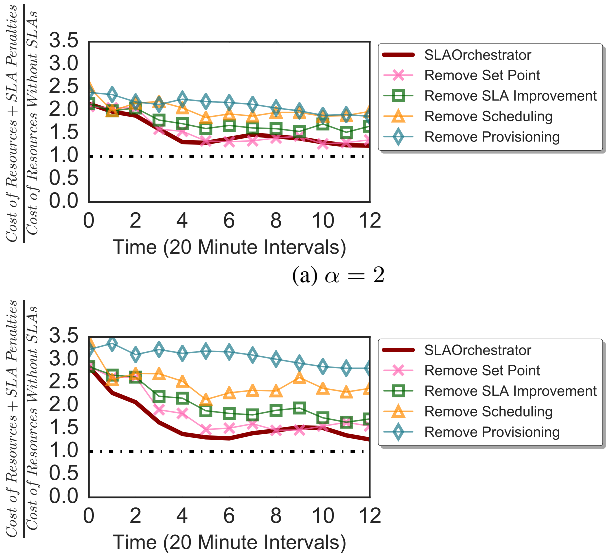
Utilization and Simulation-based Provisioning We compare the multi-tenant session costs when provisioning VMs using either the utilization-based or simulation-based approaches. We launch 100 VMs with 10 initial tenants and set the provisioning monitoring time to $20min$. Figure 8 shows the results and the other experimental parameters. For different average utilizations, Z, we show the results for the best parameter values V_{k_p} and V_{k_i}. The y-axis shows the cost per time unit, while the x-axis shows the value of the α parameter. Recall from Equation 3 that we define α as a tunable parameter that amplifies the weight of the SLA penalty compared to the resource cost. The hash pattern in each bar represents the proportion of the cost due to $Cost_R$, the cost of resources. Other costs come from SLA violations. Error bars show variance across 10 runs. As expected, the utilization-based method requires tuning depending on the α value. Simulation-based provisioning has the double-benefit of avoiding any tuning and more cost effectively provisioning shared resources compared to the utilization-based approach.

Combining Scheduling and Provisioning We now evaluate the performance of simulation-based provisioning in conjunction with various query scheduling algorithms on the initial SLA. In Figure 9, we vary alpha (x-axis) and measure the total cost compared with an Oracle query scheduler (y-axis). As a baseline, we also include a naive query scheduler, *static*, which simply schedules each query on the configuration initially purchased by the user. We also include utilization-based provisioning at $Z = .25$ (using online learning as the query scheduler). We still initialize the session with 10 tenants, but we start with a larger pool of 320 VMs, allowing enough room to have each initial tenant schedule queries on up to 32 containers. In this experiment, since we also include CMAB, we extend the session times to 180 to ensure the algorithm has more time to operate in steady state (beyond the warm-up phase).

Overall, simulation-based provisioning continues to outperform the utilization-based approach even with a less perfect scheduler. Even when penalties are high, simulation-based provisioning reduces costs by 11% and more for lower penalties. Additionally, the online learning-based scheduler yields similar costs to the Oracle scheduler (a 4% overhead). As expected, it significantly outperforms the static scheduler and CMAB when SLA penalties are expensive, with 20% cost savings. CMAB does worse because it causes more SLA violations. For small α, the CMAB approaches result in costs lower than even the Oracle scheduler. This is because the CMAB's warm-up phase initially schedules queries on all available configurations (even small configurations), which then causes the simulation approach to provision less resources. Throughout the session, resources are not added back in due to the low α value.

5.4 Dynamic Setpoint Tuning

Finally, we evaluate the benefits of dynamic setpoints together with the relative benefits of the other optimizations. Figure 10a shows the results. In the figure, we start with SLAOrchestrator as initially shown in Figure 1. We then remove optimizations one at a time in order. First, we remove the ability to use dynamic setpoints, followed by removing SLA improvements, scheduling and provisioning. We remove these optimizations to run SLAOrchestrator as a simpler multi-tenant system. To emphasize the differences between optimizations, we use $\alpha = 2$ and $\alpha = 3$. In this experiment, we start the session with 5 tenants and 80 VMs (ensuring 16 nodes per tenant, the amount they have purchased). New tenants arrive approximately every 5 minutes, and tenants finish their session after 180 minutes. As seen in the figures, removing each optimization increases the cost. This is



(a) $\alpha = 2$

(b) $\alpha = 3$

Figure 10: **Performance when disabling different SLAOrchestrator optimizations**

especially apparent for $\alpha = 3$, where SLAOrchestrator decreases the cost by up to 59% compared to the case with no optimizations.

5.5 Evaluation Discussion

The SPJ query workloads we use throughout the evaluation allow us to demonstrate a proof of concept for SLAOrchestrator. Incorporating more complex query workloads (e.g., considering aggregates and subqueries), would impact the online learning and CMAB techniques as they would require more advanced models and more extensive feature engineering than those considered in this work. Second, for a more thorough provisioning evaluation, running SLAOrchestrator against real tenant traces would help to better understand how the system would behave under more bursty workloads.

6 Related Work

Elastic Scaling for Performance Guarantees Performance guarantees have been the focus of real-time database systems [30], where the goal is to schedule queries in a fixed-size cluster and minimize deadlines. Provisioning and admission control methods have enabled OLAP and OLTP systems to make profitable choices with respect to performance guarantees [12, 11, 56], possibly postponing or even simply rejecting queries. Work by Das et. al [14], uses telemetry to determine whether to scale up containers within a single node, whereas our goal is to scale the number of containers per query. Ernest [53], CherryPick [2], Morpheus [29] and CloudScale [48] find good configurations for analytical workloads, but require representative workloads or repeated tenant usage patterns. Several systems have studied performance SLAs through dynamic resource allocation, including feedback control [33], and TIRAMOLA [31] which use reinforcement learning techniques. Others leverage decisions based on resource utilization goals [13, 15, 19, 39, 51, 57]. Tempo [52] simulates the performance of many MapReduce Resource Manager configurations to meet a global system objective, but jobs can be preempted to allow high priority tenants to finish first.

Multi-Tenant Workload Consolidation Related work addresses bad tenant packings by either finding a good initial placement strategy or dynamically migrating tenants [15, 17, 32, 34, 51, 55]. Finding a good tenant placement strategy is not the focus of our work. Instead, we focus on algorithms that help determine when to launch or turn off machines.

Query Runtime Prediction Previous work has relied on techniques to find whether a query will miss a deadline [56], build gray-box performance models [20], use historical traces of workloads [18], use benchmarks to profile resources [58], or run smaller samples of the workload [53]. Herodotou et. al. [24], assumes a previously profiled workload to predict the runtime throughout different sized clusters. Jalaparti et. al. [27] generates resource combinations given performance goals. Instead of building an analytical model, we use a model that does not require an extensive understanding of a single system. We also focus on ad-hoc queries with no prior profiles.

Provisioning In terms of provisioning, some rely on machine learning techniques such as the hill-climbing approach seen in Marcus et. al. [37], which allows machines to learn an optimal time to wait before they shut down. Neural networks for dynamic allocation [38] or dynamic provisioning [44] have also been used, but have distinct goals. One focuses on allocating resources with minimal use of electrical power while the other assumes predictable workloads.

7 Conclusion

We presented SLAOrchestrator, a new system designed to minimize the price of performance SLAs in cloud analytics systems. SLAOrchestrator uses a double learning loop that improves SLAs and resource management over time. To support the latter, the system also includes an efficient combination of elastic query scheduling and multi-tenant resource provisioning algorithms that work toward minimizing service costs. Experiments demonstrate that SLAOrchestrator dramatically reduces service costs for a common type of per-query latency SLAs.

Acknowledgments: This project was supported in part by NSF grants IIS-1247469 and IIS-1524535, and the Intel Science and Technology Center for Big Data.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, 2017. USENIX Association.

[3] Amazon RDS. http://aws.amazon.com/rds/.

[4] Amazon AWS. http://aws.amazon.com/.

[5] Amazon Elastic MapReduce (EMR). http://aws.amazon.com/elasticmapreduce/.

[6] Apache yarn. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[7] Apache Spark: Lightnight-fast cluster computing. http://spark.apache.org/.

[8] Microsoft Azure. http://azure.microsoft.com/en-us/.

[9] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[10] O. Chapelle and L. Li. An empirical evaluation of thompson sampling. In *Advances in Neural Information Processing Systems 24*. 2011.

[11] Y. Chi et al. SLA-tree: a framework for efficiently supporting sla-based decisions in cloud computing. In *Proc. of the EDBT Conf.*, pages 129–140, 2011.

[12] Y. Chi, H. J. Moon, and H. Hacigümüs. iCBS: Incremental costbased scheduling under piecewise linear SLAs. *PVLDB*, 4(9):563–574, 2011.

[13] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *Proc. of the SIGMOD Conf.*, pages 313–324, 2011.

[14] S. Das, F. Li, V. R. Narasayya, and A. C. König. Automated demand-driven resource scaling in relational database-as-a-service. In *Proceedings of the 2016 International Conference on Management of Data*, Proc. of the ACM SIGMOD International Conference on Management of Data, pages 1923–1934, New York, NY, USA, 2016. ACM.

[15] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, 2014.

[16] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proc. of the SIGMOD Conf.*, pages 301–312, 2011.

[17] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan. Characterizing tenant behavior for placement and crisis mitigation in multitenant DBMSs. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 517–528, 2013.

[18] A. D. Ferguson, P. Bodík, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 99–112, 2012.

[19] M. Fu, A. Agrawal, A. Floratou, B. Graham, A. Jorgensen, M. Li, N. Lu, K. Ramasamy, S. Rao, and C. Wang. Twitter Heron: Towards extensible streaming engines. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1165–1172, 2017.

[20] A. Gandhi, P. Dube, A. Kochut, L. Zhang, and S. Thota. Autoscaling for hadoop clusters. In *IC2E 2016*.

[21] Z. Gong, X. Gu, and J. Wilkes. PRESS: Predictive elastic resource scaling for cloud systems. In *6th IEEE/IFIP International Conference on Network and Service Management (CNSM 2010)*, Niagara Falls, Canada, 2010.

[22] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. June 2009.

[23] B. He, M. Yang, Z. Guo, R. Chen, W. Lin, B. Su, H. Wang, and L. Zhou. Wave computing in the cloud. In *Proceedings of HotOS'09: 12th Workshop on Hot Topics in Operating Systems, May 18-20, 2009, Monte Verità, Switzerland*, 2009.

[24] H. Herodotou et al. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proc. of the Second SoCC Conf.*, page 18, 2011.

[25] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 261–272, 2011.

[26] Hive. http://hadoop.apache.org/hive/.

[27] V. Jalaparti et al. Bridging the tenant-provider gap in cloud services. In *Proc. of the 3rd ACM Symp. on Cloud Computing*, page 10, 2012.

[28] P. K. Janert. *Feedback Control for Computer Systems*. O'Reilly Media, Inc., 2013.

[29] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 117–134, Berkeley, CA, USA, 2016. USENIX Association.

[30] B. Kao et al. Advances in real-time systems. chapter An Overview of Real-time Database Systems, pages 463–486. Prentice-Hall, Inc., 1995.

[31] I. Konstantinou et al. TIRAMOLA: elastic nosql provisioning through a cloud management platform. In *Proc. of the SIGMOD Conf.*, pages 725–728, 2012.

[32] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards multi-tenant performance SLOs. *IEEE Transactions on Knowledge and Data Engineering*, 26(6):1447–1463, 2014.

[33] H. Lim et al. Automated control for elastic storage. In *ICAC*, pages 1–10, 2010.

[34] Z. Liu, H. Hacigümüs, H. J. Moon, Y. Chi, and W.-P. Hsiung. PMAX: Tenant placement in multitenant databases for profit maximization. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, New York, NY, USA, 2013. ACM.

[35] K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris. Adaptive state space partitioning of markov decision processes for elastic resource management. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 191–194, 2017.

[36] R. Marcus and O. Papaemmanouil. Wisedb: A learning-based workload management advisor for cloud databases. *Proc. VLDB Endow.*, 9(10), June 2016.

[37] R. Marcus and O. Papaemmanouil. Releasing cloud databases for the chains of performance prediction models. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[38] D. Minarolli and B. Freisleben. Distributed resource allocation to virtual machines via artificial neural networks. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 490–499, 2014.

[39] V. R. Narasayya, S. Das, M. Syamala, S. Chaudhuri, F. Li, and H. Park. A demonstration of SQLVM: performance isolation in multi-tenant relational database-as-a-service. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1077–1080, 2013.

[40] P. O'Neil, E. O'Neil, and X. Chen. Star schema benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.PDF.

[41] J. Ortiz, V. T. de Almeida, and M. Balazinska. Changing the face of database cloud services with personalized service level agreements. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[42] O. Papaemmanouil. Supporting extensible performance SLAs for cloud databases. In *Proc. of the 28th ICDE Conf.*, pages 123–126, 2012.

[43] Perfenforce: A dynamic scaling engine for analytics with performance guarantees. http://myria.cs.washington.edu/publications/perfenforce_tech_report_2018.pdf.

[44] X. Qiu, M. Hedwig, and D. Neumann. *SLA Based Dynamic Provisioning of Cloud Resource in OLTP Systems*, pages 302–310. 2012.

[45] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch. A data generator for cloud-scale benchmarking. In *TPCTC'10*, pages 41–56.

[46] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan. Azure Data Lake Store: A hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 51–63, 2017.

[47] J. Rogers, O. Papaemmanouil, and U. Cetintemel. A generic auto-provisioning framework for cloud databases. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 63–68, 2010.

[48] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, page 5, 2011.

[49] Sla for azure cosmos db. https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/v1_0/.

[50] R. S. Sutton and A. G. Barto. Reinforcement learning I: Introduction, 2016.

[51] R. Taft, W. Lang, J. Duggan, A. J. Elmore, M. Stonebraker, and D. DeWitt. STeP: Scalable tenant placement for managing database-as-a-service deployments. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, 2016.

[52] Z. Tan and S. Babu. Tempo: Robust and self-tuning resource management in multi-tenant parallel databases. *Proc. VLDB Endow.*, 9(10):720–731, June 2016.

[53] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, 2016. USENIX Association.

[54] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu. The Myria big data management and analytics system and cloud services. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research*, 2017.

[55] P. Wong, Z. He, and E. Lo. Parallel analytics as a service. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 25–36, 2013.

[56] P. Xiong et al. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In *Proc. of the Second SoCC Conf.*, page 15, 2011.

[57] P. Xiong et al. SmartSLA: Cost-sensitive management of virtualized resources for CPU-bound database services. In *IEEE Transactions on Parallel and Distributed Systems*, pages 1441–1451, 2015.

[58] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 452–465, New York, NY, USA, 2017. ACM.

Spindle: Informed Memory Access Monitoring

Haojie Wang[*†], Jidong Zhai[*], Xiongchao Tang[*†], Bowen Yu[*], Xiaosong Ma[†], Wenguang Chen[*]

Abstract

Memory monitoring is of critical use in understanding applications and evaluating systems. Due to the dynamic nature in programs' memory accesses, common practice today leaves large amounts of address examination and data recording at runtime, at the cost of substantial performance overhead (and large storage time/space consumption if memory traces are collected).

Recognizing the memory access patterns available at compile time and redundancy in runtime checks, we propose a novel memory access monitoring and analysis framework, Spindle. Unlike methods delaying all checks to runtime or performing task-specific optimization at compile time, Spindle performs *common static analysis* to identify predictable memory access patterns into a compact program structure summary. Custom memory monitoring tools can then be developed on top of Spindle, leveraging the structural information extracted to dramatically reduce the amount of instrumentation that incurs heavy runtime memory address examination or recording. We implement Spindle in the popular LLVM compiler, supporting both single-thread and multi-threaded programs. Our evaluation demonstrated the effectiveness of two Spindle-based tools, performing memory bug detection and trace collection respectively, with a variety of programs. Results show that these tools are able to aggressively prune online memory monitoring processing, fulfilling desired tasks with performance overhead significantly reduced (2.54× on average for memory bug detection and over 200× on average for access tracing, over state-of-the-art solutions).

1 Introduction

Memory access behavior is crucial to understand applications and evaluate systems. They are widely monitored in system and architecture research, for memory bug or race condition detection [21, 27, 31], information flow tracking [16, 30], large-scale system optimization [35, 36, 42], and memory system design [14, 17, 20].

Memory access monitoring and tracing need to obtain and check/record memory addresses visited by a program and this process is quite expensive. Even given complete source-level information, much of the relevant information regarding locations to be accessed at runtime is not available at compile time. For example, it is common that during static analysis, we see a heap object accessed repeately in a loop, but does not have any of the parameters needed to perform our desired examination or tracing: where the object is allocated, how large it is, or how many iterations there are in a particular execution of the loop. As a result, existing memory checking tools mostly delay the checking/transcribing of such memory addresses to execution time, with associated instructions instrumented to perform task-specific processing. Such runtime processing brings substantial performance overhead (typically bringing 2× or more application slowdown [5, 33] for online memory access checking and much higher for memory trace collection [6, 22, 26]).

However, there are important information not well utilized at compile time. Even with actual locations, sizes, branch taken decisions, or loop iteration counts unknown, we still see patterns in memory accesses. In particular, accesses to large objects are not isolated events that have to be verified or recorded individually at runtime. Instead, they form groups with highly similar (often identical) behaviors and *relative* displacement in locations visited given plainly in the code. The processing tasks that are delayed to execution time often perform the same checking or recording on individual members of such large groups of highly homogeneous accesses. In addition, the memory access patterns recognizable during static analysis summarize *common structural information* useful to many memory checking/tracing tasks.

Based on these observations, we propose Spindle,

[*]Department of Computer Science and Technology, Tsinghua University {wanghaoj15, txc13, yubw15}@mails.tsinghua.edu.cn, {zhaijidong, cwg}@mail.tsinghua.edu.cn

[†]Qatar Computing Research Institute, HBKU {whaojie, txiongchao, xma}@qf.org.qa

a new platform that facilitates hybrid static+dynamic analysis for efficient memory monitoring. It leverages common static analysis to identify from the target program the source of redundancy in runtime memory address examination. By summarizing groups of memory accesses with statically identified program structures, such compact intermediate analysis results can be passed to Spindle-based tools, to further perform task-specific analysis and code instrumentation. The regular/predictable patterns contained in Spindle-distilled structural information allow diverse types of memory access checking more efficiently: by ***computing*** rather than ***collecting*** memory accesses whenever possible, even when certain examination has to be conducted at runtime, it can be elevated from instruction to object granularity, with the amount of instrumentation dramatically pruned.

We implement Spindle on top of the open-source LLVM compiler infrastructure [10]. On top of it, we implement two proof-of-concept custom tools, a memory bug detector (S-Detector) and a memory trace collector (S-Tracer), that leverage the common structural information extracted by Spindle to optimize their specific memory access monitoring tasks.

We evaluated Spindle and the aforementioned custom tools with popular benchmarks (NPB, SPEC CPU2006, Graph500, and PARSEC) and open-source applications covering areas such as machine learning, key-value store, and text processing. Results show that S-Detector can reduce the amount of instrumentation by 64% on average using Spindle static analysis results, allowing runtime overhead reduction of up to 30.25× (2.54× on average) over the Google AddressSanitizer [33]. S-Tracer, meanwhile, reduces the trace collection time overhead by up to over 500× (228× on average) over the polular PIN tool [22], and cuts the trace storage space overhead by up to over 10000× (248× on average).

Spindle is publicly available at `https://github.com/thu-pacman/Spindle`.

2 Overview

2.1 Spindle Framework

Spindle is designed as a hybrid memory monitoring framework. Its main module performs static analysis to extract program structures relevant to memory accesses. Such structural information allows Spindle to obtain regular or predictable patterns in memory accesses. Different Spindle-based tools utilize these patterns in different ways, with the common goal of reducing the amount of instrumentation that leads to costly runtime check or information collection.

Figure 1 gives the overall structure of Spindle, along with sample memory monitoring tools implemented on top of it. To use Spindle-based tools, end-users only have to compile their application source code with the Spindle-enhanced LLVM modules, whose output then goes through tool-specific analysis and instrumentation. More specifically, the common static analysis performed by Spindle will generate a highly compact *Memory Access Skeleton (MAS)*, describing the structured, predictable memory access components.

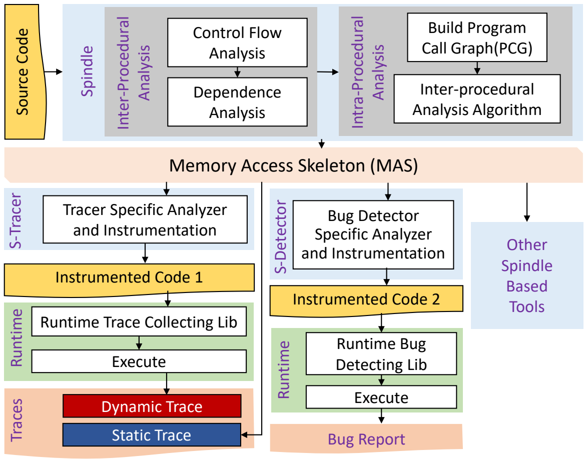

Source Code
Spindle
Inter-Procedural Analysis
Control Flow Analysis
Dependence Analysis
Intra-Procedural Analysis
Build Program Call Graph(PCG)
Inter-procedural Analysis Algorithm
Memory Access Skeleton (MAS)
S-Tracer
Tracer Specific Analyzer and Instrumentation
Instrumented Code 1
Runtime
Runtime Trace Collecting Lib
Execute
Traces
Dynamic Trace
Static Trace
S-Detector
Bug Detector Specific Analyzer and Instrumentation
Instrumented Code 2
Runtime
Runtime Bug Detecting Lib
Execute
Bug Report
Other Spindle Based Tools

Figure 1: Spindle overview

Spindle tool developers write their own analyzer, which uses MAS to optimize their code instrumentation, aggressively pruning unnecessary or redundant runtime checks or monitoring data collection. In general, such task-specific tools enable *computing* groups of memory addresses visited before or after program executions, to avoid *examining* individual memory accesses at runtime. As illustrated in Figure 1, each of such Spindle-based tools (the memory bug detector S-Detector and memory trace collector S-Tracer in this case) will generate its own instrumented application code. As our results will show, for typical applications, the majority of memory accesses are computable given a small amount of runtime information, leading to dramatic reduction of instrumentation and runtime collection.

End-users then execute their tool-instrumented applications, with again task-specific runtime libraries linked. The instrumented code conducts runtime processing to perform the desired form of memory access monitoring, such as bug or race condition detection, security check, or memory trace collection. The runtime libraries capture dynamic information to fill in parameters (such as the starting address of an array or the actual iteration count of a loop) to instantiate the Spindle MAS and complete the memory monitoring tasks. In addition, all the "unpredictable" memory access components, identified by Spindle at compile time as input-dependent, are monitored/recorded in the traditional manner.

Spindle's static analysis workflow to produce MAS is further divided into multiple stages, performing intra-procedural analysis, inter-procedural analysis, as well as tool specific analysis and instrumentation. During the intra-procedural stage, Spindle analyzes the program control flow graph and finds out the dependence among memory access instructions. The dependence checking is then expanded across functions in inter-procedural analysis.

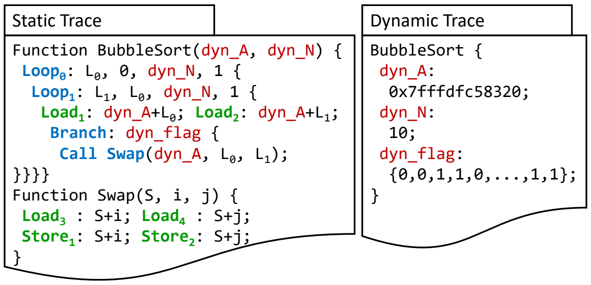
One limitation of the current Spindle framework is that it requires source level information of target programs. As this work is a proof-of-concept study, also considering the current trend of open-source software adoption [9, 41], our evaluation uses applications with source code available. Spindle can potentially work without source code though: it starts with LLVM IR and can therefore employ open-source tools such as Fcd [7] or McSema [37] to translate binary codes into IR. In our future work we are however more interested in direct static analysis, performing tasks such as loop and dependency detection on binaries.

2.2 Sample Input/Output: Memory Trace Collector

```
void BubbleSort(int *A, int N){
    for (int i = 0; i < N; ++i){
        for (int j = i+1; j < N; ++j){
            bool flag = (A[i] > A[j]);
            if (flag) {
                Swap(A, i, j);
}}}}

void Swap(int *S, int i, int j) {
    int tmp = S[i];
    S[i] = S[j];
    S[j] = tmp;
}
```

Figure 2: Sample bubble sort program



Static Trace

```
Function BubbleSort(dyn_A, dyn_N) {
 Loop_0: L_0, 0, dyn_N, 1 {
  Loop_1: L_1, L_0, dyn_N, 1 {
   Load_1: dyn_A+L_0; Load_2: dyn_A+L_1;
    Branch: dyn_flag {
     Call Swap(dyn_A, L_0, L_1);
}}}}
Function Swap(S, i, j) {
 Load_3 : S+i; Load_4 : S+j;
 Store_1: S+i; Store_2: S+j;
}
```

Dynamic Trace

```
BubbleSort {
 dyn_A:
  0x7fffdfc58320;
 dyn_N:
  10;
 dyn_flag:
  {0,0,1,1,0,...,1,1};
}
```

Figure 3: Memory traces of the bubble sort program

We take S-Tracer, our Spindle-based trace collector, as an example to give a more concrete picture of Spindle's working. Suppose the application to be monitored is the bubble sort program listed in Figure 2. S-Tracer's output, given in Figure 3, is a complete yet compressed memory access trace, consisting of its MAS (coupled with corresponding dynamic parameters) and dynamic traces collected in the conventional manner.

In the static trace, we list out the structure of the program, including the control flow, the memory accesses pattern and the call graph. There are information items that cannot be determined during static analysis, such as the base address of array A and its size N, which is also the final value of loop induction variables i and j , as well as the value of flag, which is data-dependent and determines the control flow of this program. The "Instrumented code 1" shown in Figure 1 records these missing values at executing time, which compose the dynamic trace shown on the right.

This new trace format, though slightly more complex than traditionally traces, is often orders of magnitude smaller. A straightforward post-processor can easily take S-Tracer traces and restore the traditional full traces. More practically, an S-Tracer trace driver performing similar decompression can be prepended to typical memory trace consumers, to enable fast replay without involving large trace files or slow I/O.

3 Static Analysis

3.1 Intra-procedural Analysis

During this first step, Spindle extracts a program's per-function control structure to identify memory accesses whose traces can be computed and hence can be (mostly) skipped in dynamic instrumentation.

3.1.1 Extracting Program Control Structure

A program's memory access patterns (or the lack thereof) are closely associated to its control flows. It is not surprising that it shares a similar structure with the program's control flow graph (CFG). Therefore we call this graph M-CFG. Unlike traditional control flow graphs, M-CFG records only instructions containing memory references (rather than the entire basic block), program control structures (loops and branches), and function calls. For loops and branches, we need to record related variables, such as loop boundaries and branch conditions.

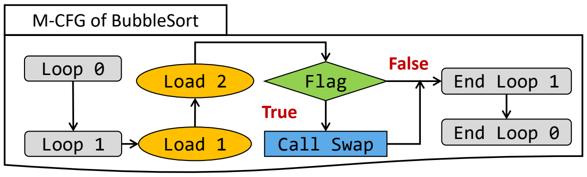


Figure 4: The M-CFG for the function BubbleSort

With M-CFG, memory access instructions are embedded within program basic control structures, as illustrated in Figure 4 for the aforementioned BubbleSort function (Figure 2). Here the M-CFG records a nested loop containing two memory accesses and a branch with a function call. Subsection 3.1.2 discusses dependence analysis regarding memory access instructions and identification of computable memory accesses, while Section 3.2 discusses as handling of function calls.

3.1.2 Building Memory Dependence Trees

In Spindle, we classify all memory accesses into either *computable* or *non-computable* types. The computable accesses can have traces computed based on the static trace, with the help of little or no dynamic information; the non-computable ones, on the other hand, need to fall back to traditional instrumentation and runtime tracing.

For such classification, we build a *memory dependence tree* for each memory access instruction. It records data dependence between a specific memory access instruction and its related variables. The tree is rooted at the memory address accessed, with non-leaf nodes denoting operators between variables such as addition or multiplication and leaf nodes denoting variables in the program. Edges hence intuitively denote dependence.

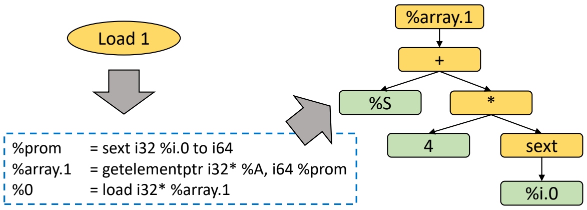
Below we list the types of leaf nodes in memory dependence trees:

- *Constant value*: value determined at compile time
- *Base memory address*: start address for continuously allocated memory region (such as an array), with value acquired at compile time for global or static variables, and at runtime for dynamically allocated variables.
- *Function parameter*: value determined at either compile time or runtime (see Section 3.2)
- *Data-dependent variable*: value dependent on data not predictable at compile time – to be collected at runtime
- *Function return value*: value collected at runtime
- *Loop induction variable*: variable regularly updated at each loop iteration, value determined at compile time or runtime

Algorithm 1 Algorithm of building memory dependence tree

```
input: A worklist $WL[A]$. Predefined Leaf types: $Type$
output: memory dependence tree: $T(A)$
Insert a root note $r$ to $T(A)$
while $WL[A] \neq \phi$ do
    Remove an item $v_1$ from $WL[A]$
    if $v_1 \notin Type$ then
        for $v_2 \in UD(v_1)$ do
            if $v_2 \in Type$ then
                Insert a leaf node $v_2$
                Insert an edge from $v_1$ to $v_2$
            else
                Insert an operator node in $v_2$ to $T(A)$
                Add all variables used in $v_2$ to $WL[A]$
    else
        Insert a leaf node $v_1$ to $v_1$ to $T(A)$
        Insert an edge from $r$ to $v_1$ to $T(A)$
return $T(A)$
```

The memory dependence tree is built by performing a backward data flow analysis at compile time. Specifically, for each memory access, we start from the variable storing this memory address and traverse its use-define data structure, which describes the relation between the definition and use of each variable, to identify all the variables and operators affecting it. This traversal is an iterative process that stops when all the leaf nodes are categorized into one of the types listed above. We give the worklist algorithm (Algorithm 1) that performs such backward data flow analysis with, where we repeatedly variables storing memory addresses into the worklist $WL(A)$ and iteratively find all the related variables through the use-define structure $UD(v)$, till the worklist becomes empty.



```
%prom      = sext i32 %i.0 to i64
%array.1   = getelementptr i32* %A, i64 %prom
%0         = load i32* %array.1
```

Figure 5: Sample memory dependence tree

Figure 5 shows a group of instructions (generated from the source code in Figure 2) and the memory dependence tree corresponding to the variable `%array.1` in the last line. Here `getelementptr` is an instruction that calculates the address of an aggregate data structure (where an addition operation is implied) and does not access memory. We omit certain arguments for this instruction for simplicity. `sext` performs type casting. As to the leaf nodes, `%A` is an array base address, `4` is a constant value, and `%i.0` is a loop induction variable.

Such a dependence tree allows us to approach the central task of Spindle: ***computable memory access identification***. This is done by analyzing the types of the leaf nodes in the memory dependence tree. Intuitively, a memory access is computable if the leaf nodes of its dependence tree are either constants (trivial) or loop induction variables (computable by replicating computation performed in the original program using initial plus final values, collected at compile time or runtime). The M-CFG and the memory access dependence trees, preserving control flows, data dependencies, and operations to facilitate such replication, can be viewed as a form of *program pruning* that only retains computation relevant to memory address calculation. By replacing each memory instruction of the M-CFG with its dependence tree, we obtain a single graph representing main memory access patterns for a single function. Note that such dependence analysis naturally handles aliases.

3.2 Inter-procedural Analysis

At the end of the intra-procedural analysis, we have a memory dependence tree for every memory access within each function. Below we describe how Spindle analyzes memory address dependence across functions.

The core idea here is to propagate function arguments *plus their dependence* from the caller to the callee, and replace all the function parameters of the dependence trees in the callee with actual parameters. For this, we first build a program call graph (PCG), on which we subsequently perform top-down inter-procedural analysis. Algorithm 2 gives the detailed process.

Algorithm 2 The algorithm of inter-procedural analysis

```
input: The dependence trees for each procedure p
input: The program call graph (PCG)
Change ← True
/* Top-Down inter-procedural analysis */
while (Change == True) do
    Change ← False
    for all procedure p in Pre-Order over PCG do
        for all dependence trees d in p do
            if A leaf node l of d is a function's parameter then
                Replace l with its actual parameter
                Change ← True
```

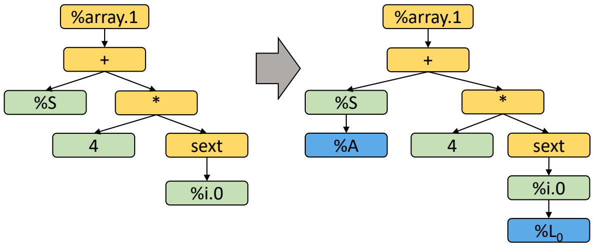


Figure 6: Transformation of dependence tree

Figure 6 illustrates the transformation a dependence tree in function `Swap` (Figure 2) undergoes during inter-procedural analysis. After intra-procedural analysis, the dependence tree for the load instruction `Load`$_3$ of function `Swap` has two leaf nodes that are function parameters, which cannot be analyzed then as the variables `%S` and `%i.0` are undetermined. Within inter-procedural analysis, these two nodes are replaced with their actual parameters, a base address `%A` and a loop induction variable `%i.0` Now the dependence tree rooted at `%array.1` is computable.

For function calls forming a loop in PCG, such as recursive calls, currently we do not perform parameter replacement for any function in this loop during our inter-procedural analysis, as when these functions terminate is typically data-dependent.

3.3 Special Cases and Complications

Index arrays If a memory dependence tree has data-dependent variables as its leaf nodes, normally we consider it non-computable. However, we still have chance to extract regular patterns. Index array is an important case of such data-dependent variables, storing "links" to other data structures, as explained below.

```
for (j=0; j<i; j++){
    for (k=0; k<m; k++)
        sum += delta * z[colidx[k]]
        //colidx is index array to z
    r[k] = d
}
```

Figure 7: NPB CG code with index array `colidx`

Figure 7 gives a simplified version of a code snippet from NPB CG [2], where the array `z` is repeatedly accessed via the index array `colidx`, which cannot be determined at compile time. However, we find that in many programs (including here) the index array itself is not modified across multiple iterations of accesses. Therefore, there is still significant room for finding repeated access patterns and removing redundancy.

To this end, Spindle performs the following extra evaluation during its static analysis. First, it compares the size of index array and its total access count. If the latter is larger, we only need to record the content of the index array and *compute* the memory accesses accordingly rather than instrumenting them at runtime. Such evaluation needs to be repeated if the content of this index array is changed, of course. This is the case with the example given in Figure 7, where the total memory access count for the index array `colidx` is `i*m` and greater than the size of `colidx`. Thus at runtime we only need to record its content at the beginning of this nested loop and the base address of array `z`. Combining such information and memory dependence tree, we can compute all the memory access locations.

Multi-threaded programs The discussion so far has been focused on analyzing single-thread programs. However, Spindle's methodology can also be easily applied to multi-threaded applications. Spindle is thread-safe and we perform the same static analysis as for single-thread programs, except that we also mark the point where a new thread is created and record relevant parameter values. With parallel executions, during dynamic memory monitoring (discussed in the next section), the current thread ID would be easily fetched along with information such as loop iteration count and branch taken, which allows us to distinguish runtime information collected by different threads. Note that certain techniques need to be augmented to handle multi-threaded executions. E.g., the array index technique (Section 3.3) needs to be protected by additional check, as an array could be modified by another thread.

Again, with addresses or values that cannot be determined at compile time, such as shared objects or branches affected by other threads, we fall back to runtime instrumentation. So typical SPMD codes will share the same static MAS, to be supplemented by per-thread or even per-process runtime information, making Spindle even more appealing in efficiency and scalability. If significant amount of output is generated, such as with memory trace collection, Spindle allows users to have the option to look at a single-thread's memory accesses or correlating accesses from all threads (though trace interleaving is a separate topic that requires further study.)

For example, with *pthread*, Spindle instruments `pthread_create` to record where a new thread is created. During multi-threaded execution, the appropriate thread ID is recorded for each function. Thus we know which thread the dynamic information collected by Spindle belongs to, therefore can apply per-thread static analysis, similar to that in single-thread executions.

4 Spindle-based Runtime Monitoring

This section illustrates how Spindle's static analysis results can be used to reduce runtime instrumentation. We first describe common runtime information to be obtained through instrumentation, then present two samples of Spindle-based tool design, for memory bug detection and memory trace collection, respectively.

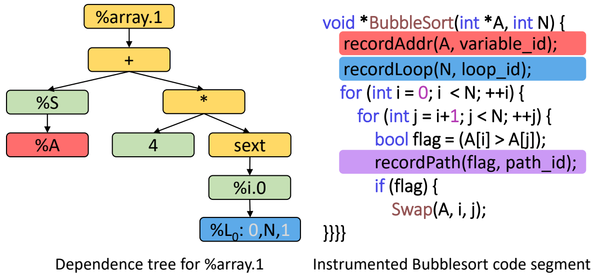
4.1 Runtime information collection

During program runs, Spindle's static memory access skeleton is supplemented by information not available at compile time. Generally, three cases require instrumentation: control structures, input-dependent variables, and non-computable memory accesses:

Control structures Spindle needs to record the initial values of all the loop induction variables and the loop iteration count if they are unknown at compilation time. Moreover, for a loop with multiple exit points, we need to instrument each exit point to track where the loop exits. Similarly, for conditional branches in MAS, we need to record their taken statuses to track taken paths.

Input dependent variables For input dependent variables, runtime information is necessary but certain static analysis can indeed reduce runtime overhead. For instance, the address of a dynamically allocated memory region can be obtained at runtime by collecting actual values. An optimization in Spindle is that we do not instrument every instruction that references input dependent variables, but only where they are defined, initialized, or updated. E.g., for a global variable needed by the analysis, it leverages static analysis to only record its initial value at the beginning of the program, and then again upon its updates.

Non-computable memory accesses For non-computable memory accesses (as mentioned in Subsection 3.1.2), we fall back to conventional dynamic monitoring/instrumentation.



```
void *BubbleSort(int *A, int N) {
  recordAddr(A, variable_id);
  recordLoop(N, loop_id);
  for (int i = 0; i  < N; ++i) {
    for (int j = i+1; j < N; ++j) {
      bool flag = (A[i] > A[j]);
      recordPath(flag, path_id);
      if (flag) {
        Swap(A, i, j);
}}}}
```

Dependence tree for %array.1 Instrumented Bubblesort code segment

Figure 8: Sample runtime information collection

Figure 8 shows an example of runtime information collection for the BubbleSort routine discussed earlier in Section 2.2. The left side gives the dependence tree of the variable %array.1 in function Swap, where undetermined address %A and loop L$_0$'s induction variable %N need to be collected at runtime. Note that L$_0$'s initial index value (0) and increment (1) can be determined at compile time. The right side lists the instrumented BubbleSort code. Here Spindle automatically instruments three memory accesses by inserting the highlighted statements (for A, N, and the branch related flag, which falls out of the dependence tree shown). variable_id, loop_id, and path_id are also automatically generated by Spindle for its runtime library to find the appropriate static structures.

4.2 Spindle-based tool developing

Spindle's performs automatic code instrumentation for runtime information collection, based on its static analysis. To build a memory monitoring tool on top of Spindle, users only need to supply additional codes using its API to perform custom analysis, as to be illustrated below. Our two sample tools, S-Detector and S-Tracer, each takes under 500 lines of code to implement both compile-time analysis and runtime library.

4.2.1 Memory Bug Detector (S-Detector)

Memory bugs, such as buffer overflow, use after free, and use before initialization, may cause severe runtime errors or failures, especially with programming languages like C and C++. There have been a series of tools, software- or hardware-based, developed to detect memory bugs at compile-time or runtime. Among them, Memchecker [39] uses hardware support for memory access monitoring and debugging and is therefore fast (only 2.7% performance overhead for SPECCPU 2000). Such special-purpose hardware is nevertheless not yet adopted by general processors. ARCHER [43] relies on static analysis only, so is faced with the difficult trade-off between accuracy (false positives) and soundness (false negatives), like other static tools. A recent, state-of-the-art tool is AddressSanitizer (ASan) [33], an industrial-strength memory bug detection tool developed by Google and now built into the LLVM compiler. ASan inserts memory checking instructions (such as out-of-bound array accesses) into programs at compile time, then uses shadow memory [25] for fast runtime checking. Despite well implemented and highly tuned, ASan still introduces 2–3× slowdown to SPEC programs.

In this work, we present S-Detector, a memory bug detector that leverages Spindle-gathered static information to eliminate unnecessary instrumentation to facilitate efficient online memory checking. Our proof-of-concept implementation of S-Detector can currently detect invalid accesses (e.g., out-of-bound array access and use after free) and memory leaks (dynamically allocated objects remaining unfreed upon program termination).

With Spindle's MAS, S-Detector is aware of a program's groups of memory accesses and therefore able to perform checking at a coarser granule. E.g., with dynamically allocated arrays, even when neither the starting address (*base*) or size (*bound*) is known at compile time, its accesses are given as relative to these two values and can therefore be checked for out-of-bound bugs at compile time. With existing tools like ASan, however, such checks are delayed till runtime and repeated at every memory acesses.

Therefore, S-Detector performs aggressive *memory check pruning* by proactively conducting compile-time access analysis and replacing instruction-level checks by object-level ones. Only for accesses labeled "non-computable" by Spindle, S-Detector falls back to traditional instrumentation. Below, we illustrate S-Detector's memory check pruning with two sample scenarios, both contained in the same code snippet from SPEC CPU2006 `mcf` (Figure 9).

```
1  while (pos - 1 && red_cost >
2              (cost_t)new[pos/2-1].flow){
3      new[pos-1].tail = new[pos/2-1].tail;
4      new[pos-1].head = new[pos/2-1].head;
5      // Three more accesses to struct members
6      // of new[pos-1] and new[pos/2-1].
7      pos = pos/2;
8      new[pos-1].tail = tail;
9      // Four more accesses to struct members
10     // of new[pos-1].
11 }
```

Figure 9: Sample code from SPEC CPU 2006 `mcf`

In-structure accesses This sample code references an array of structures (`new`), issuing multiple accesses to members of its elements. In this case, assisted with Spindle-extracted MAS, all access targets can be represented as `addr = struct_base + constant_offset`. Once S-Detector finds that the constant `offset` is valid for this struct, i.e., `offset<struct_size`, it only needs to determine if this structure element itself is valid at runtime, i.e., the memory range `[struct_base, struct_base + struct_size)` is a valid range. This groups the per-member access checks to per-element checks (validating structure elements like `new[pos-1]` and `new[pox/2-1]`) and significantly reduces the amount of instrumentation.

In-loop accesses Given the `while` loop in the same sample code, Spindle records the following information for its loop induction variable `pos`: its initial and final values (denoted here as `pos_init` and `pos_final`), as well as the operation used to update it across iterations (divided by 2 at Line 7). Based on the MAS, S-Detector can easily infer the offset range of array `new`'s access to be within [`pos_end/2-1`, `pos_init-1`]. In addition, it records array `new`'s size in bytes (`new_size`) and the size of `new`'s elements (`struct_size`). Aside from quick checks to ensure that the object has been allocated and not freed yet, S-Detector verifies that

$$(\texttt{pos\_init} - 1) * \texttt{struct\_size} < \texttt{new\_size} \quad (1)$$

and

$$\texttt{pos\_end}/2 - 1 \geq 0 \quad (2)$$

Actually inequality (2) is guaranteed by the loop's exit condition, so S-Detector only needs to check (1). Even when none of these four parameter values is available at compile time, S-Detector only needs to perform a *one-time*, *object-level* check at runtime, for array object accesses within this `while` loop.

Combining the structure- and loop-level pruning described above, S-Detector can eliminate all per-instruction memory checks on accesses of the `new` object in the sample code, performing at most one single run-time check instead.

4.2.2 Memory Trace Collector (S-Tracer)

Complete, detailed memory access traces allow diverse analysis and faithful benchmarking or simulation tests. However, their colletion is expensive, both in time and space. Existing tools like PIN [22], Valgrind [26], and DynamoRIO [6] produce memory trace output of daunting sizes, due to the high frequency of memory accesses in typical program executions. It is common for several seconds' execution to generate hundreds of GBs, sometimes even over one TB, of memory traces using any of the existing tools. Large memory trace size not only introduces large overhead for underlying trace storage and various trace-based analysis tools, but also affects the performance of the original programs. For example, PIN introduces an average slowdown of 38× for SPEC INT programs to perform memory analysis [38]. In addition, large traces bring back the I/O bottleneck during replay time, slowing down trace-driven simulations. Such limitations make it less and less practical for existing memory tracing tools to measure significant portions of modern data-intensive applications.

We present S-Tracer, a memory trace collection tool based on Spindle. With the static information that provided by Spindle, S-Tracer can generate highly compressed memory access traces with much lower runtime overhead than traditional tracing tools using dynamic instrumentation. At runtime, S-Tracer couples the Spindle-extracted MAS with dynamically collected information mentioned earlier in this section. The result would be a pair of static and dynamic traces, as illustrated in Figure 2 and Figure 3.

Our discussion below focuses on specific challenges due to the limitation of using LLVM IR, where we propose several techniques to generate approximate but fairly accurate traces.

Register spilling Since Spindle performs its static analysis in the LLVM IR level, where local scalar variables are usually represented as register variables, it is difficult for our approach to capture the stack memory accesses caused by register spilling in the final binary code. Considering the small footprint of register variables even with spilling, we implement typical register allocators used in the compiler backend for Spindle at the IR level, to calculate register spilling. Based on our experiments, our approach is able to achieve the similar statistical behavior of stack accesses as by traditional tracing tools.

Implicit memory accesses with function calls Function calls can also generate stack memory operations, not explicitly described in IR and hence not captured by our intra- and inter-procedural analysis. There are two categories of such accesses. For the caller, it has to write into stack the return address, the contents of registers to be used, and function parameters (with x86_64, the first 6 parameters are put in registers while the others in stack). For the callee, upon returning it has to read from stack the return address of the caller, the content of register `EBP` (for 32-bit systems) or `RBP` (for 64-bit systems), and the content of saved registers. To handle this, we again write a simple simulator to generate these memory accesses.

Dynamically linked libraries Since Spindle performs source code analysis, for calls to functions in dynamically linked libraries, we cannot capture their memory accesses in the IR level and have to fall back again to traditional dynamic instrumentation. As an optimization, we adopt a hybrid approach, by using dynamic instrumentation to collect the relative memory traces within such functions, along with their base stack addresses within the dynamic library. When a program calls such a function, we can then calculate new memory accesses based on the new base stack address.

5 Evaluation

In this section, we demonstrate the effectiveness of Spindle with the aforementioned two sample tools built on top of its static analysis framework: S-Detector for online memory bug detection and S-Tracer for full memory access trace collection.

We compare S-Detector with the state-of-the-art memory bug detector, ASan [33] by Google. In our experiments, S-Detector and ASan do the same checks: use after free, heap buffer overflow, stack buffer overflow, global buffer overflow, and memory leaks. Note that ASan does support additional checks (use after return, use after scope, and initialization order bugs), which need to be explicitly enabled by certain compiler options. Our tests used the default compiler options and we performed extra verification to confirm that these additional checks were disabled in all of our ASan experiments.

For S-Tracer, we show that it produces orders of magnitude smaller trace output, and thus lower overhead, by omitting redundant information. To validate its correctness, we also compare its decompressed trace with trace generated by PIN, a widely used dynamic tool.

5.1 Experiment Setup

Test platform We evaluate Spindle on a server with Intel Xeon E7-8890 v3 processors (running CentOS 7.1), 128GB of DDR3 memory, and 1TB SATA-2 hard disk. For memory bug detection, the tests use mandatory options to enable ASan and DrMem. For memory trace collection, we record each memory access in a 16-byte entry, 8 bytes for memory address and another 8 bytes for access type (read/write) and access size.

Test programs Currently, Spindle fully supports C and partially supports C++ and Fortran. For memory bug detection, we follow the practice of previously published tools and use 11 C programs from SPEC CPU 2006 [1]: `400.perlbench`, `401.bzip2`, `403.gcc`, `429.mcf`, `433.milc`, `445.gobmk`, `456.hmmer`, `458.sjeng`, `464.h264ref`, `470.lbm`, and `482.sphinx3`. The program `998.specrand` is omitted as it has too few memory accesses. Using these common test programs, we not only can compare the tools' runtime overhead, but also their effectiveness of capturing known bugs.

For memory trace collection, we use the popular NPB parallel benchmark suite [2] as codes with mostly regular memory accesses, plus SPEC `429.mcf` as a memory-intensive, non-numerical program. We also sample from modern data-intensive and irregular datacenter applications: (1) the Breadth First Search (`BFS`) component of the Graph500 Benchmark [11], a representative graph application with input-dependent memory accesses, (2) a convolutional neural network for digit recognition (`MNIST`) [29], (3) `kissdb`, a key-value store [18], and (4) `Fido`, a lightweight, modular machine learning library [8]. Finally, for multi-threaded applications, we test 3 programs from the PARSEC suite [4] covering different application domains: `streamcluster` (stream processing), `freqmine` (data mining), and `blackscholes` (PDE solving), plus one MapReduce [23]-style program performing `word count`, denoted as `SC`, `FM`, `BS` and `WC` respectively.

5.2 Spindle Compilation Overhead

Before we get to the tool use cases, we first assess the extra overhead brought by Spindle's static analysis. Table 1 summarizes this compilation overhead for evaluated programs, as well as their original compilation time and code size. In general, the Spindle compilation overhead only composes a small fraction of the original LLVM compilation cost (2% to 35%, average at 10%). We consider such one-time static analysis overhead negligible, considering the significant savings in the much larger runtime checking/tracing cost.

Table 1: Spindle compilation overhead

Program	Extra	Original	Code size	Program	Extra	Original	Code size
BT	0.260s	4.170s	232KB	perlbench	4.662s	23.036s	4418KB
CG	0.084s	0.651s	35KB	bzip2	0.053s	2.828s	239KB
EP	0.043s	0.493s	10KB	gcc	1.596s	66.729s	13777KB
FT	0.098s	0.908s	40KB	mcf	0.028s	0.694s	62KB
IS	0.049s	0.427s	25KB	milc	0.360s	3.899s	458KB
LU	0.225s	3.260s	244KB	gobmk	1.444s	16.921s	239KB
MG	0.161s	0.984s	43KB	hmmer	0.924s	8.773s	1126KB
SP	0.228s	2.320s	164KB	sjeng	0.270s	2.521s	298KB
BFS	0.704s	4.142s	852KB	h264ref	2.556s	15.268s	1656KB
MNIST	0.399s	1.138s	4KB	lbm	0.076s	0.906s	44KB
kissdb	0.092s	1.835s	16KB	sphinx3	0.304s	5.106s	767KB
FM	0.535s	7.760s	112KB	Fido	1.051s	9.287s	160KB
SC	0.159s	3.407s	80KB	BS	0.068s	2.250s	15KB
WC	0.054s	1.324s	19KB				

5.3 S-Detector for Memory bug detection

S-Detector runtime overhead We compare S-Detector with two popular memory bug detection tools: Google's AddressSanitizer (ASan) [33] and DynamoRIO [6]-based Dr. Memory (DrMem) [5]. To examine the benefits of instrumentation pruning based on Spindle's static analysis, we test two versions of S-Detector: *SD-All*, a baseline version that instruments all memory accesses, and *SD-Opt*, after check pruning.

On bug detection results, S-Detector captures most of the common SPEC bugs reported by DrMem and ASan, plus additional memory leaks (dynamically allocated objects not freed by program termination) that are verified by our manual code examination.

Figure 10 shows the runtime overhead of ASan, SD-All and SD-Opt, in percentage of the original program execution time. As DrMem is much heavier than others (for most programs over 10× slowdown), we omit its results from the figure for clarity. ASan is an industrial-strength tool, whose streamlined implementation delivers lower overhead than SD-All (geometric mean of overhead at 66% by the former vs. 184% by the latter), both with similar amount of instrumentation. SD-Opt, however, overcomes its slower checking implementation and brings down runtime overhead to geometric mean of 26%. Except for two out of 11 cases (`bzip2` and `h264ref`), SD-Opt reduces overhead from ASan, by up to 30.25× (`sphinx3`). We give more detailed discussion of these special cases later.

Spindle-enabled instrumentation pruning To take a closer look, we examine the amount of checks avoided by Spindle's static analysis. Figure 11 gives the percentage of eliminated memory checks, from SD-All to SD-Opt. On average, Spindle enables S-Detector to cut runtime memory checks by 64%, lowering its performance overhead consequently. The check and overhead reduction level depends on several factors, such as the amount of irregular/unpredictable memory accesses (Amdahl's Law), the overall intensiveness of memory accesses, and control flow behavior. Below we give more detailed results and analysis via several case studies.

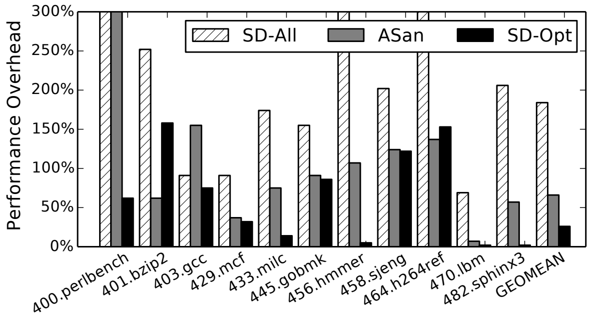


Figure 10: Overhead comparison (bars over 300% truncated)

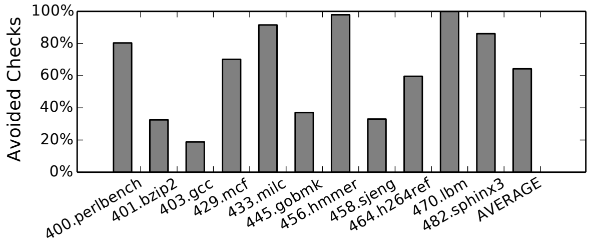


Figure 11: Reduction in runtime memory checks

`lbm`, `hmmer`, `milc`: These are the best cases among tested. Function-level profiling shows that the vast majority of their execution time and most of their memory accesses are spent within loops, where Spindle analysis allows S-Detector to apply the loop-level check presented in Section 4.2.1, replacing the per-access checks performed by ASan and DrMem. As a result, these three programs have 99%, 97%, and 91% of memory checks removed by S-Detector, respectively. Such instrumentation pruning then lowers S-Detector's runtime overhead, e.g., to 5% for `hmmer`, vs. ASan's 107%.

`gcc`: this compiler program is inherently input-dependent and as a result, has the lowest reduction by S-Detector in memory checks (19%). Interestingly, though its execution does spend most time within Spindle-identified loop structures, most of its loops are found to run only a few iterations, limiting the benefit of S-Detector's loop-level static checks. However, in this case even SD-All is faster than ASan. Follow-up measurements reveal that S-Detector's shadow memory implementation, though less efficient in general, offers better spatial locality than ASan's. With `gcc` accessed memory areas being particularly spread out, ASan's runtime check harms its locality, bringing the LLC miss rate from the original 1.3% to 5.9%, while S-Detector retains the original caching performance.

`bzip2`: this compression/decompression program is also input-dependent. Profiling reveals a performance hot-spot in sorting, with many branches whose taken status relies on input data. Even with 32% of runtime memory checks pruned, the less efficient instrumentation of

S-Detector brought overall higher overhead than ASan, 158% vs. 62%.

Despite such worst cases, the overall strong performance of S-Detector indicates that its Spindle-based static analysis, if adopted by highly-tuned, mature tools like ASan, may lead to even lower runtime overhead.

5.4 S-Tracer for Memory Trace Collection

Result Trace Verification Next, we evaluate S-Tracer, comparing it with the widely used PIN tool [22] for memory tracing. We first validate the correctness of its memory trace generation. Note that Spindle is based on compile-time instrumentation while traditional tools like PIN use runtime instrumentation. The two systems run application programs within different frameworks, each with different components (such as dynamic libraries), which may in turn alter the absolute locations of memory objects. Therefore, one would not expect them to generate identical trace sequences.

Recognizing such limitations, we first check the output trace size. We compare the size of PIN's trace with full traces recovered from Spindle's output, in the same format. The Spindle recovered trace has the similar volume to PIN's, with relative difference between 0.5% and 6% (median at 3.2%). Additional examination reveals that such discrepancies stem from the aforementioned inaccuracy caused by Spindle's approximation of stack accesses and register spilling. Though amounting for up to a few percent of the overall trace entries, affected accesses are typically localized to a very small footprint and hardly impact the overall memory access behavior.

We then validate the Spindle-generated heap memory access sequence. We examine trace fidelity by performing more detailed trace alignment and checking difference in heap access sequences. For each access on heap, we break it into a pair: (`object`, `offset`), since for each execution the dynamically allocated `object`'s `base` is different but the `offset` remains constant. We use `Linux diff` tool to compare S-Detector's heap trace and PIN's and find that overall, S-Tracer generates heap traces close to PIN's (relative difference ratio between 0.0% and 4.7%, median at 1.5%).

In the worst case, S-Tracer could generate an overall 5.9% difference in total trace size and 4.7% difference ratio on heap accesses, mostly attributed to stack accesses (more influenced by register allocation) and register spilling. Below we test this worst case, `BFS`, using a cache simulator, to (1) demonstrate a use case of our fast and large-capacity memory tracing and (2) provide a validation for trace fidelity. The test uses a simple trace-driven tool that simulates an 8-way set-associative cache with 64-byte cache line, and two replacement algorithms (LRU and FIFO). We validate simulation results using S-Tracer traces against that using PIN's, at varied cache sizes (including typical L2 and LLC sizes). Figure 12 shows that S-Tracer output achieves almost identical outcome as the PIN trace in miss ratio, across different combinations of cache size and replacement strategies.

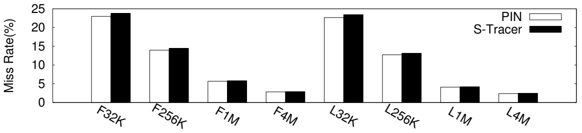


Figure 12: The cache miss rate of `BFS` in a trace-driven simulator. F means FIFO algorithm, L means LRU algorithm. The size means the cache size we simulate.

Trace Size Reduction Next we assess S-Tracer's gain in tracing time/space efficiency. Figure 13 shows a comparison of the trace size generated by S-Tracer and PIN, *in log scale*, for 13 single-thread and 4 multi-threaded programs. Truncated bars are from programs whose PIN traces exceed our 1TB storage capacity (`BT`, `EP`, `LU`, `SP` of Class A). For S-Tracer, the trace size includes both the static and dynamic components.

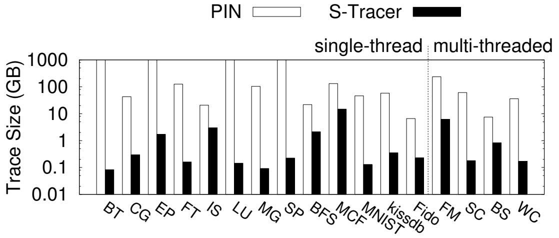


Figure 13: Trace size comparison

As expected, S-Tracer achieves orders of magnitude reduction in trace size from the PIN baseline. For programs dominated by regular memory accesses, like most of the programs in NPB benchmark, `MNIST`, `kissdb`, `streamcluster`, and `wordcount`, it reduces trace size by more than 100×. For the four NPB benchmarks where PIN exceeds the 1TB storage space, S-Tracer generates traces sized at 85MB-1.71GB. Even for the less regular programs, such as `BFS` and `freqmine`, Spindle brings considerable trace size reduction. In the worst case (`IS`, integer sorting), a 6.93× reduction is achieved.

We also evaluated compressing PIN's trace with a naive alternative, `gzip`, which ended up producing orders of magnitude larger traces than S-Tracer does. Besides, generating then compressing traces is much more expensive than Spindle-based approach, online or offline.

Runtime Tracing Overhead Reduction To evaluate the runtime overhead of trace collection, Figure 14 shows the slowdown factor (left axis, in *log scale*), calculated by dividing the execution time with tracing by the original time, for S-Tracer and PIN.

As expected, the online overhead difference is dramatic. In the 13 programs that PIN can complete tracing (full trace size under 1TB disk space), the average slowdown is 502× (and up to over 2000×), while S-

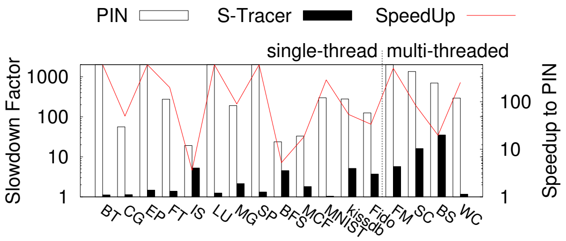


Figure 14: Application slowdown by S-Tracer and PIN with I/O (left) and S-Tracer speedup over PIN (right)

Tracer brings that of 6.5× on average (and up to 35.2×), making full trace collection/storage much more affordable. Across the applications, S-Tracer reduces slowdown from PIN by a factor of 61× on average.

Though we do not have space to show the no-I/O results, the savings there are still significant. For the 17 test programs, PIN introduces an average slowdown of 70.1× (and up to 384×), while S-Tracer brings that of 4.5× on average (and up to 33×). Across the applications, S-Tracer reduces slowdown from PIN by a factor of 17.9× on average. The reason is that Spindle allows S-Tracer to perform far less dynamic instrumentation, and an application's relative time overhead is highly correlated to its dynamic trace generation rate.

6 Related Work

Using Static Analysis to Assist Runtime Checking This group of work is closest to Spindle in approach. In particular, GreenArray [24] is an LLVM-based tool that analyzes the value range of index variables as well as the boundary of memory regions at compile time, to eliminate unnecessary runtime memory check. Spindle is different in that (1) its static analysis performs much more than inferring variables' value range, allowing complete computation of their value by iteration and full trace collection, and (2) the static skeleton it produces enables more types of and much more aggressive pruning in runtime checking, judging by reported GreenArray performance relative to AddressSanitizer.

Abstract Execution (AE) [19] produces a target-event-specific program slice, to be coupled by a "schema compiler" with runtime collected information and executed again for analysis or trace collection. Spindle, instead, records static trace at compile time, which is directly utilized during the target programs (production) execution.

On utilizing static information to assist trace collection, Cypress [44] uses hybrid static-dynamic analysis for parallel programs' communication trace compression. There are also techniques that perform static binary rewriting/instrumentation [32] or regular-expression-based memory access pattern construction for memory layout transformation [15]. However, none of these approaches is able to gather enough static structural information to enable versatile runtime monitoring/tracing as Spindle does.

Also, logical connectives proposed for relational analyses between input and output memory states [13] may be used by Spindle to further reduce instrumentation.

Monitoring/Tracing overhead reduction Prior work has explored reducing monitoring or tracing overhead in other ways. MemTrace [28] performs lightweight memory tracing of unmodified binary applications by translating 32-bit codes to 64-bit codes, which is fast but limits its application to running 32-bit programs on 64-bit machines. Among sampling-based methods, Vetter [40] evaluates techniques for analyzing communication activity in large-scale distributed applications. RACEZ [34] uses hardware performance monitoring units to sample memory accesses at runtime, and then uses the collected memory access trace for offline data-race detection. However, such low-overhead methods lose important information, such as temporal order of operations, or miss detection targets.

Finally, Bao et al. [3, 12] adopt a DIMM-snooping hardware mechanism to collect virtual memory reference traces. This hardware solution indeed minimizes collection overhead, but is rather costly and only catches memory accesses missed by on-chip caches.

7 Conclusion and Future Work

This paper presents Spindle, a versatile memory monitoring framework that performs detailed static analysis to extract program structures, allowing different types of static and dynamic techniques to *compute* rather than *collect* memory accesses whenever possible. Our development and experiments confirm that there are abundant redundancy and regularity in memory accesses, even for applications perceived as more irregular and data-dependent. By identifying predictable memory access behaviors at compile time and supplementing statically obtained memory access skeletons with runtime information, we can dramatically reduce the amount of online checking (for purposes like bug or race detection) or data collection (for purposes like memory access pattern analysis or memory tracing).

Acknowledgment

We thank all reviewers for their insightful comments and our shepherd Samira Khan for her timely guidance. We also thank colleagues from both the Tsinghua University PACMAN group and the QCRI Distributed Systems group, for their valuable feedback and suggestions. This work is supported in part by the National Key R&D Program of China (Grant No. 2017YFA0604500), National Natural Science Foundation of China (Grant No. 61722208, 61472201), Tsinghua University Initiative Scientific Research Program (20151080407). Jidong Zhai is the corresponding author of this paper.

References

[1] SPEC CPU 2006. https://www.spec.org/cpu2006/.

[2] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow. *The NAS Parallel Benchmarks 2.0*. NAS Systems Division, NASA Ames Research Center, Moffett Field, CA, 1995.

[3] Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei Xu. Hmtt: A platform independent full-system memory trace monitoring system. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '08, pages 229–240. ACM, 2008.

[4] The PARSEC benchmark. http://parsec.cs.princeton.edu/.

[5] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223, Los Alamitos, CA, USA, 2011.

[6] Derek L Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[7] The fcd tool. https://github.com/zneak/fcd/.

[8] The fido library. http://fidoproject.github.io/.

[9] Brian Fitzgerald, Jay P Kesan, Barbara Russo, Maha Shaikh, and Giancarlo Succi. *Adopting Open Source Software*. MIT Press, 2011.

[10] The LLVM Compiler Framework. http://llvm.org.

[11] Graph500. http://www.graph500.org/.

[12] Yongbing Huang, Licheng Chen, Zehan Cui, Yuan Ruan, Yungang Bao, Mingyu Chen, and Ninghui Sun. Hmtt: A hybrid hardware/software tracing system for bridging the dram access trace's semantic gap. *ACM Trans. Archit. Code Optim.*, 11(1):7:1–7:25, 2014.

[13] Hugo Illous, Matthieu Lemerre, and Xavier Rival. A relational shape abstract domain. In *NASA Formal Methods Symposium*, pages 212–229. Springer, 2017.

[14] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *SIGMETRICS'07*, pages 25–36. ACM, 2007.

[15] Jinseong Jeon, Keoncheol Shin, and Hwansoo Han. Layout transformations for heap objects using static access patterns. In *Proceedings of the 16th International Conference on Compiler Construction*, CC'07, pages 187–201, 2007.

[16] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM SIGPLAN Notices*, volume 47, pages 121–132. ACM, 2012.

[17] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Micro*, pages 65–76, 2010.

[18] The kissdb program. https://github.com/adamierymenko/kissdb.git.

[19] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software Practice Experience*, 20(12):1241–1258, November 1990.

[20] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT'12*, pages 367–376. ACM, 2012.

[21] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 37–48. ACM, 2006.

[22] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200. ACM, 2005.

[23] The mapreduce program. https://github.com/sysprog21/mapreduce.git.

[24] Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. Validation of memory accesses through symbolic analyses. In *ACM SIGPLAN Notices*, volume 49, pages 791–809. ACM, 2014.

[25] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[26] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100. ACM, 2007.

[27] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36. ACM, 2009.

[28] Mathias Payer, Enrico Kravina, and Thomas R Gross. Lightweight memory tracing. In *USENIX Annual Technical Conference*, pages 115–126, 2013.

[29] The CNN program. `https://github.com/preimmortal/CNN.git`.

[30] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *ACM Sigplan Notices*, pages 245–258. ACM, 2009.

[31] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pattabiraman. Detecting and tolerating asymmetric races. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 173–184. ACM, 2009.

[32] Amitabha Roy, Steven Hand, and Tim Harris. Hybrid binary rewriting for memory access instrumentation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 227–238. ACM, 2011.

[33] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[34] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, and Robert Hundt. Racez: A lightweight and non-invasive race detection tool for production applications. In *ICSE*, pages 401–410, 2011.

[35] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. pages 45–57, 2002.

[36] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *SC*, pages 1–17, 2002.

[37] The McSema tool. `https://github.com/trailofbits/mcsema/`.

[38] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing dynamic binary instrumentation overhead. In *WBIA Workshop at ASPLOS*, 2006.

[39] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 273–284. IEEE, 2007.

[40] Jeffrey Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 240–250. ACM, 2002.

[41] Georg Von Krogh and Eric Von Hippel. The promise of research on open source software. *Management science*, 52(7):975–983, 2006.

[42] Shasha Wen, Milind Chabbi, and Xu Liu. Redspy: Exploring value locality in software. In *ASPLOS*, pages 47–61. ACM, 2017.

[43] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *ACM SIGSOFT Software Engineering Notes*, 28(5):327–336, 2003.

[44] Jidong Zhai, Jianfei Hu, Xiongchao Tang, Xiaosong Ma, and Wenguang Chen. Cypress: Combining static and dynamic analysis for top-down communication trace compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'14, pages 143–153, 2014.

Touchstone: Generating Enormous Query-Aware Test Databases

Yuming Li[1], Rong Zhang[1]*, Xiaoyan Yang[2], Zhenjie Zhang[2], Aoying Zhou[1]
[1]*East China Normal University, {liyuming@stu, rzhang@sei, ayzhou@dase}.ecnu.edu.cn*
[2]*Singapore R&D, Yitu Technology Ltd., {xiaoyan.yang, zhenjie.zhang}@yitu-inc.com*

Abstract

Query-aware synthetic data generation is an essential and highly challenging task, important for database management system (DBMS) testing, database application testing and application-driven benchmarking. Prior studies on query-aware data generation suffer common problems of limited parallelization, poor scalability, and excessive memory consumption, making these systems unsatisfactory to terabyte scale data generation. In order to fill the gap between the existing data generation techniques and the emerging demands of enormous query-aware test databases, we design and implement our new data generator, called *Touchstone*. *Touchstone* adopts the random sampling algorithm instantiating the query parameters and the new data generation schema generating the test database, to achieve fully parallel data generation, linear scalability and austere memory consumption. Our experimental results show that *Touchstone* consistently outperforms the state-of-the-art solution on TPC-H workload by a 1000× speedup without sacrificing accuracy.

1 Introduction

The applications of query-aware data generators include DBMS testing, database application testing and application-driven benchmarking [5, 15]. For example, during the database selection and performance optimization, the internal databases in production are hard to be shared for performance testing due to the privacy considerations, so we need to generate synthetic databases with the similar workload characteristics of the target queries. A bulk of existing data generators, e.g., [12, 11, 4, 20], generate test databases independent of the test queries, which only consider the data distribution of inter- and intra-attribute. They fail to guarantee the similar workload characteristics of the test queries, therefore it's difficult to match the overheads of the query execution engine for real world workloads. A number of other studies, e.g., [6, 14, 5, 15], attempt to build query-aware data generators. But the performance of the state-of-the-art solution MyBenchmark [15] still remains far from satisfactory, due to the lack of parallelization, scalability and memory usage control, as well as the narrow support of non-equi-join workload. In order to generate the enormous query-aware test databases, we design and implement *Touchstone*, a new query-aware data generator, based on the following design principles:

Full Parallelization: With the explosive growth of data volume in the industrial applications, the database system is expected to support storage and access services for terabyte or even petabyte scale data. So the synthetic data generator must be fully parallel for generating such extremely large test databases.
Linear Scalability: The single machine has been far from meeting the requirements of generating large test databases, and the data scales may be unbelievably big for the future applications, therefore the data generator needs to be well scalable to multiple nodes and data size.
Austere Memory Consumption: When generating the synthetic database for multiple queries, memory could easily be the bottleneck, because massive information is maintained by the data generator in order to guarantee the dependencies among columns. The memory usage needs to be carefully controlled and minimized.

Since [6, 14, 5, 15] are closest to the target of this work, we list the following key insufficiencies of these studies for elaborating the necessity of proposing *Touchstone*. In particular, all of these approaches do not support fully parallel data generation in a distributed environment due to the primitive data generation algorithms over the huge shared intermediate state, limiting the efficiency of data generation over target size at terabytes. Moreover, their memory consumptions, e.g., symbolic databases of QAGen [6], constrained bipartite graphs of WAGen [14] and MyBenchmark [15], caching referenced tables for generating foreign keys of DCGen [5], strongly depend on the size of generation outputs. Once the memory is insufficient to host the complete intermediate state, vast computational resources are wasted on disk I/O operations. In addition, one key advantage of our work is the support of non-equi-join workload, which is important for real world applications but not supported by any of the existing approaches.

In query-aware data generation, we need to handle the extremely complicated dependencies among columns which are caused by the complex workload characteristics specified on the target test queries, as well as the data characteristics specified on the columns. *Touchstone* achieves fully parallel data generation, linear scalability and austere memory consumption for supporting the generation of enormous query-aware test databases. There are two core techniques employed by *Touchstone* beneath the accomplishments of all above enticing features. Firstly, *Touchstone* employs a completely new

*Rong Zhang is the corresponding author.

query instantiation scheme adopting the random sampling algorithm, which supports a large and useful class of queries. Secondly, *Touchstone* is equipped with a new data generation scheme using the constraint chain data structure, which easily enables thread-level parallelization on multiple nodes. In summary, *Touchstone* is a scalable query-aware data generator with a wide support to complex queries of analytical applications, and achieves a 1000× performance improvement against the state-of-the-art work MyBenchmark [15].

2 Preliminaries

2.1 Problem Formulation

The input of *Touchstone* includes database schema H, data characteristics D and workload characteristics W, as illustrated in Figure 1. H defines data types of columns, primary key and foreign key constraints. In Figure 1, there are three tables R, S, and T. For example, table S has 20 tuples and three columns. Data characteristics D of columns are defined in a meta table, in which the user defines the percentage of *Null* values, the domain of the column, the cardinality of unique values and the average length and maximum length for varchar typed columns. In our example, the user expects to see 5 unique values on column $R.r_2$ in the domain [0, 10], and 8 different strings with an average length of 20 and a maximum length of 100 for column $T.t_3$ with 20% *Null* values. Workload characteristics W are represented by a set of parameterized queries which are annotated with several cardinality constraints. In Figure 1, our sample input consists of four parameterized queries, i.e., $Q = \{Q_1, Q_2, Q_3, Q_4\}$. These four queries contain 11 variable parameters, i.e., $P = \{P_1, P_2, ..., P_{11}\}$. Each filter/join operator in the queries is associated with a size constraint, defining the expected cardinality of the processing outcomes. Therefore, there are 14 filter/join operators and corresponding 14 cardinality constraints in our example, i.e., $C = \{c_1, c_2, ..., c_{14}\}$. Our target is to generate the three tables and instantiate all the variable query parameters. In the following, we formulate the definition of cardinality constraints.

Definition 1 ***Cardinality Constraint:*** *Given a filter (σ) or join ($\bowtie$) operator, a cardinality constraint c is denoted as a triplet c = [Q, p, s], where Q indicates the involving query, p gives the predicate on the incoming tuples, and s is the expected cardinality of operator outcomes.*

The cardinality constraint c_1 in Figure 1, for example, is written as [Q_1, $R.r_2 < P_1$, 4], indicating that the operator with predicate $R.r_2 < P_1$ in query Q_1 is expected to output 4 tuples. For conjunctive and disjunctive operators, their cardinality constraints can be split to multiple cardinality constraints for each basic predicate using standard probability theory. These cardinality constraints generally characterize the computational workload of query processing engines, because the computational overhead mainly depends on the size of the data in processing. This hypothesis is verified in our experimental evaluations.

While the focus of cardinality constraints is on filter and join operators, *Touchstone* also supports complex queries with other operators, including aggregation, groupby and orderby. For example, the query Q_2 in Figure 1 applies groupby operator on $T.t_3$ and summation operator on $S.s_3$ over the grouped tuples. The cardinality of the output tuples from these operators, however, is mostly determined, if it does not contain a having clause. And such operators are usually engaged on the top of query execution tree, hence the output result cardinalities generally do not affect the total computational cost of query processing. Based on these observations, it is unnecessary to pose explicit cardinality constraints over these operators [14, 5] in *Touchstone*.

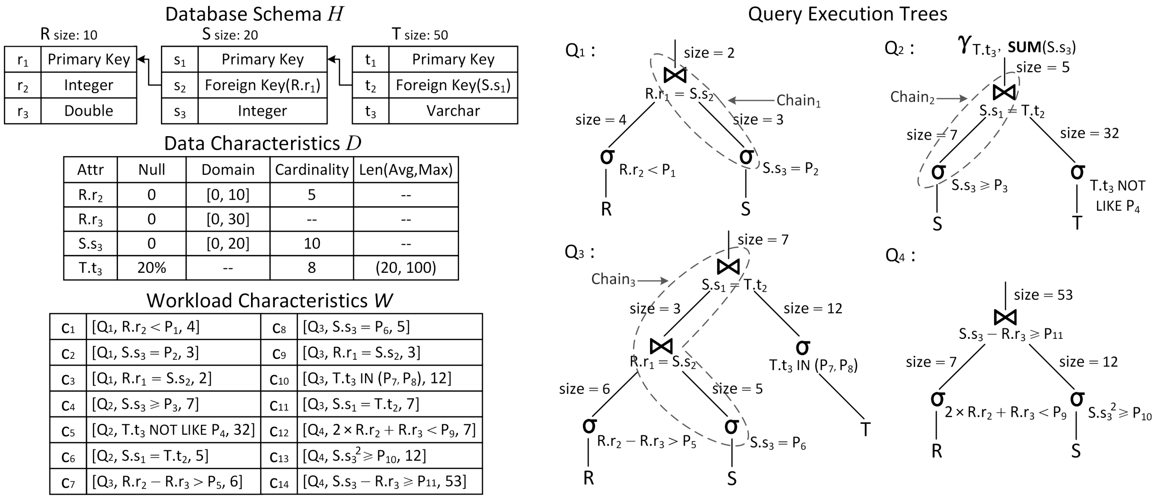
Based on the target operators (filter or join) and the predicates with equality or non-equality conditional expressions, we divide the cardinality constraints into four types, i.e., $C = C^{\sigma}_{=} \cup C^{\sigma}_{\neq} \cup C^{\bowtie}_{=} \cup C^{\bowtie}_{\neq}$. Accordingly, we classify the example constraints in Figure 1 as $C^{\sigma}_{=} = \{c_2, c_5, c_8, c_{10}\}$[1], $C^{\sigma}_{\neq} = \{c_1, c_4, c_7, c_{12}, c_{13}\}$, $C^{\bowtie}_{=} = \{c_3, c_6, c_9, c_{11}\}$ and $C^{\bowtie}_{\neq} = \{c_{14}\}$. Following the common practice in [5, 24, 25], the equi-join operator is *always* applied on the pair of primary and foreign keys.

Then we formulate the problem of query-aware data generation as follows.

Definition 2 ***Query-Aware Data Generation Problem:*** *Given the input database schema H, data characteristic D and workload characteristics W, the objective of data generation is to generate a database instance (DBI) and instantiated queries, such that 1) the data in the tables strictly follows the specified data characteristics D; 2) the variable parameters in the queries are appropriately instantiated; and 3) the executions of the instantiated queries on the generated DBI produce exactly the expected output cardinality specified by workload characteristics W on each operator.*

While the general solution to query-aware data generation problem is NP-hard [21], we aim to design a data generator, by relaxing the third target in the definition above. Specifically, the output DBI is expected to perform as closely as the cardinality constraints in C. Given the actual/expected cardinalities of processing outputs, i.e., $\{\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n\}$, corresponding to constraints on the queries in $C = \{c_1, c_2, \ldots, c_n\}$, we aim to minimize the global relative error $\frac{\sum_{c_i \in C} |c_i.s - \hat{s}_i|}{\sum_{c_i \in C} c_i.s}$. Even if the user specified workload in W contains conflicted constraints,

[1] If the relational operator in a selection predicate belongs to {=, *!=*, *in*, *not in*, *like*, *not like*}, then the corresponding cardinality constraint is classified to $C^{\sigma}_{=}$.



Database Schema H

R size: 10

r_1	Primary Key
r_2	Integer
r_3	Double

S size: 20

s_1	Primary Key
s_2	Foreign Key(R.r_1)
s_3	Integer

T size: 50

t_1	Primary Key
t_2	Foreign Key(S.s_1)
t_3	Varchar

Data Characteristics D

Attr	Null	Domain	Cardinality	Len(Avg,Max)
R.r_2	0	[0, 10]	5	--
R.r_3	0	[0, 30]	--	--
S.s_3	0	[0, 20]	10	--
T.t_3	20%	--	8	(20, 100)

Workload Characteristics W

c_1	[Q_1, R.$r_2 < P_1$, 4]	c_8	[Q_3, S.$s_3 = P_6$, 5]
c_2	[Q_1, S.$s_3 = P_2$, 3]	c_9	[Q_3, R.r_1 = S.s_2, 3]
c_3	[Q_1, R.r_1 = S.s_2, 2]	c_{10}	[Q_3, T.t_3 IN (P_7, P_8), 12]
c_4	[Q_2, S.$s_3 \geqslant P_3$, 7]	c_{11}	[Q_3, S.s_1 = T.t_2, 7]
c_5	[Q_2, T.t_3 NOT LIKE P_4, 32]	c_{12}	[Q_4, $2 \times$ R.r_2 + R.$r_3 < P_9$, 7]
c_6	[Q_2, S.s_1 = T.t_2, 5]	c_{13}	[Q_4, S.$s_3^2 \geqslant P_{10}$, 12]
c_7	[Q_3, R.r_2 − R.$r_3 > P_5$, 6]	c_{14}	[Q_4, S.s_3 − R.$r_3 \geqslant P_{11}$, 53]

Figure 1: Example inputs of database schema, data characteristics and workload characteristics to *Touchstone*

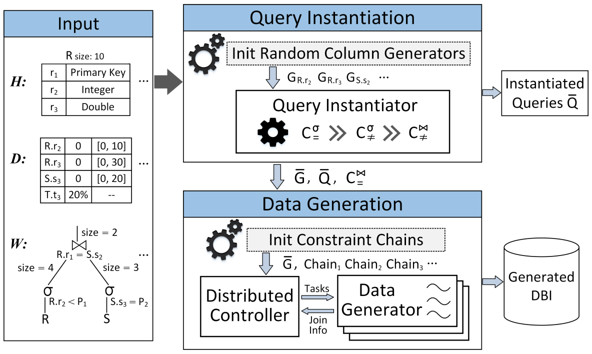


Figure 2: The overall architecture of *Touchstone*

Touchstone still attempts to generate a DBI with the best effort.

2.2 Overview of Touchstone

The infrastructure of *Touchstone* is divided into two components, which are responsible for query instantiation and data generation respectively, as shown in Figure 2.

Query Instantiation: Given the inputs including database schema H, data characteristics D, *Touchstone* builds a group of random column generators for non-key columns, denoted by G, each G_i in which corresponds to a column of the target tables. Given the input workload characteristics W, *Touchstone* instantiates the parameterized queries by adjusting the related column generators if necessary and choosing appropriate values for the variable parameters in the predicates of $c \in C^{\sigma}_{=} \cup C^{\sigma}_{\neq} \cup C^{\bowtie}_{\neq}$. The instantiated queries $\bar{Q}$ are output to the users for reference, while the queries $\bar{Q}$ and the adjusted column generators $\bar{G}$ are fed into the data generation component. The technical details are available in Section 3.

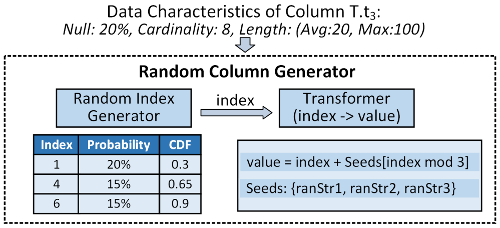
Data Generation: Given the inputs including instantiated queries $\bar{Q}$ and constraints over the equi-join operators $C^{\bowtie}_{=}$ specified in W, *Touchstone* decomposes the query trees annotated with constraints into constraint chains, in order to decouple the dependencies among columns, especially for primary-foreign-key pairs. Data generation component generally deploys the data generators over a distributed platform. The random column generators and constraint chains are distributed to all data generators for independent and parallel tuple generation. The technical details are available in Section 4.

2.3 Random Column Generator

The basic elements of *Touchstone* system are a group of random column generators $G = \{G_1, G_2, \ldots, G_n\}$, which determine the data distributions of all non-key columns to be generated. A random column generator G_i in G is capable of generating values for the specified column, while meeting the required data characteristics *in expectation*. In the following, we give the detailed description of the random column generator.

A random column generator G_i contains two parts, a random index generator and a value transformer as shown in Figure 3. In the random index generator, the output index domain is the integers from 0 to $n-1$ while n is the specified cardinality of unique values in corresponding column. Given an index, the transformer deterministically maps it to a value in the specified domain of the column. We adopt different transformers based on the type of the column. For numeric types, e.g., *Integer*, we simply pick up a linear function which uniformly maps the index to the value domain. For string types, e.g., *Varchar*, there are some seed strings pregenerated randomly, which satisfy the specified length requirements. We first select a seed string based on the input index as shown in Figure 3, and then concatenate the index and the selected seed string as the output value. This approach allows us to easily control the cardinality of string typed columns with tiny memory consumption.

To manipulate the distribution of the column values, there is a probability table in the random index generator.


Data Characteristics of Column T.t3:
Null: 20%, Cardinality: 8, Length: (Avg:20, Max:100)
Random Column Generator
Random Index Generator
index
Transformer (index -> value)

Index	Probability	CDF
1	20%	0.3
4	15%	0.65
6	15%	0.9

value = index + Seeds[index mod 3]

Seeds: {ranStr1, ranStr2, ranStr3}

Figure 3: An example generator for column $T.t_3$

The probability table consists of a number of entries and each entry corresponds to an index. Specifically, each entry in the table (k_i, p_i, c_i) specifies an index k_i, the probability p_i of occurrence, and the cumulative probability c_i for all indexes no larger than k_i. In order to save the memory space, we compress the table by keeping only the entries with non-uniform probabilities. If an index does not appear in the probability table, its probability is automatically set by the uniform probability. The entries in the table are ordered by k_i. In Figure 3, we present an example of random column generator designed for column $T.t_3$ from example inputs in Figure 1. The specified data characteristics request this column to contain 8 unique strings with average length 20 and maximum length 100. In the result generator, based on the indexes in $[0,7]$ generated by random index generator, the transformer outputs random strings with the desired lengths, at probabilities $\{p_{0,2,3,5,7} = 0.1, p_1 = 0.2, p_{4,6} = 0.15\}$. The details of probability assignment will be discussed in Section 3.

Value Generation: Given the random column generator, firstly, a *Null* value is output with the probability of the specified percentage. If *Null* value is not chosen, the index generator picks up an index based on the probabilities by running binary search over CDF (c_i) in the probability table with a random real number in $(0,1]$, and the transformer outputs the corresponding column value.

3 Query Instantiation

There are two general objectives in query instantiation, targeting to 1) construct the random column generators for each non-key column in the tables; and 2) find concrete values for the variable parameters in the queries.

Generally speaking, the query instantiation component is responsible for handling three types of constraints, i.e., $C^{\sigma}_{=}$, $C^{\sigma}_{\neq}$ and $C^{\bowtie}_{\neq}$. Note that the fourth type of constraints $C^{\bowtie}_{=}$ involves matching between primary and foreign keys, which is taken care of by the data generation process at runtime. In Algorithm 1, we list the general workflow of query instantiation. The algorithm iteratively adjusts the data distribution adopted by the random column generators and the concrete values of the variable parameters, in order to meet the constraints as much as possible. The distribution adjustment on the column generator is accomplished by inserting entries in its probability table. In each iteration, the algorithm reinitializes the column generators (line 3) such that there is no entry in the probability table, namely the probabilities of candidate values are uniform. The algorithm then attempts to adjust the column generators in $\bar{G}$ and the concrete values of the variable parameters in queries $\bar{Q}$ (lines 4-11). Specifically, it firstly adjusts the column generators and instantiates the variable parameters based on the equality constraints over filters (lines 4-6). It then follows to revise the variable parameters in the queries in order to meet the non-equality constraints on filter and join operators (lines 7-11). The details of the adjustment on column generators and the parameter instantiation are presented in the following subsections. The algorithm outputs the new (adjusted) generators $\bar{G}$ and the instantiated queries $\bar{Q}$, when the global relative error for all constraints is within the specified threshold θ or the number of iterations reaches its maximum I.

Algorithm 1 Query instantiation

```
Input: Initial generators $G$, input queries $Q$, error threshold $\theta$ and maximum number of iterations $I$
Output: New generators $\bar{G}$ and instantiated queries $\bar{Q}$
 1: Initialize $\bar{Q} \leftarrow Q$
 2: for all iteration $i = 1$ to $I$ do
 3:     Initialize $\bar{G} \leftarrow G$
 4:     for all constraint $c \in C^{\sigma}_{=}$ do
 5:         Adjust the generator in $\bar{G}$ for the column within $c$
 6:         Instantiate the corresponding parameter in $\bar{Q}$
 7:     for all $c \in C^{\sigma}_{\neq}$ do
 8:         Instantiate the corresponding parameter in $\bar{Q}$
 9:     for all $c \in C^{\bowtie}_{\neq}$ do
10:         Obtain constraints from all descendant nodes
11:         Instantiate the corresponding parameter in $\bar{Q}$
12:     Calculate the global relative error $e$
13:     if $e \leq \theta$ then return $\bar{G}$ and $\bar{Q}$
14: return $\bar{G}$ and $\bar{Q}$ (historical best solution with minimum $e$)
```

In the rest of the section, we discuss the processing strategies for these three types of constraints respectively.

Filters with Equality Constraint always involve a single non-key column at a time like the workloads of standard benchmarks. Given all these equality constraints on the filter operators, i.e., $C^{\sigma}_{=}$, the system groups the constraints according to the involved column. In our running example in Figure 1, there are four such constraints $C^{\sigma}_{=} = \{c_2, c_5, c_8, c_{10}\}$, among which, c_2 and c_8 target column $S.s_3$, and c_5 and c_{10} target column $T.t_3$. Note that all relational operators in equality constraints are handled by treating them as '='. For example, $c_5 = [Q_2, T.t_3 \text{ NOT LIKE } P_4, 32] \Rightarrow [Q_2, T.t_3 \text{ LIKE } P_4, 8] \Rightarrow [Q_2, T.t_3 = P_4, 8]$.

The processing strategy for equality constraints on filters runs in three steps. Firstly, the algorithm randomly selects an index and obtains the corresponding value from the transformer of the column generator, for instantiating each variable parameter in the equality con-

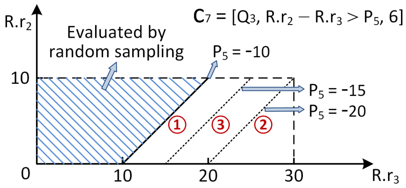


Figure 4: An example of parameter searching procedure for constraint c_7. Given the predicate in c_7, our algorithm attempts to cut the space by revising the parameter P_5. For a concrete P_5, the expected number of tuples meeting the predicate is evaluated by the random sampling algorithm. The best value for P_5 is returned, after the binary search identifies the optimal value at desired precision or reaches the maximum iterations.

straints. Secondly, the algorithm updates the occurrence probability of the selected index in the column generator by inserting an entry in the probability table, in order to meet the required intermediate result cardinality. Whether the filter is the leaf node of the query execution tree or not, the probability of the inserted entry is calculated as $\frac{s_{out}}{s_{in}}$, where s_{in} is the size of input tuples and s_{out} is the expected size of output tuples. After the above two steps for all equality constraints, the algorithm calculates the cumulative probabilities in the probability table of adjusted column generators. In Figure 3, there are three entries in the probability table for generating data with the distribution that satisfies the constraints c_5 and c_{10}. For example, the entry with index 1 is inserted for instantiating parameter P_4 in the predicate of c_5, while the two entries with indexes 4 and 6 are inserted for instantiating parameters P_7 and P_8 in the predicate of c_{10}.

Suppose there are k variable parameters in the equality constraints over filters. The total complexity of the processing strategy is $O(k \log k)$, because the algorithm only needs to instantiate the parameters one by one, and accordingly it inserts an entry into the probability table in order of selected index for each parameter instantiation.

Filters with Non-Equality Constraint could involve multiple non-key columns. In Figure 1, some constraints, e.g., $c_1 = [Q_1, R.r_2 < P_1, 4]$ and $c_4 = [Q_2, S.s_3 \geq P_3, 7]$, apply on one column only, while other constraints, e.g., $c_7 = [Q_3, R.r_2 - R.r_3 > P_5, 6]$ and $c_{12} = [Q_4, 2 \times R.r_2 + R.r_3 < P_9, 7]$, involve more than one column with more complex mathematical operators. Our underlying strategy handling these non-equality constraints is to find the concrete parameters generating the best matching output cardinalities against the constraints, based on the data distributions adopted by the random column generators.

Since the cardinality of tuples satisfying the constraints is monotonic with the growth of the variable parameter, it suffices to run a binary search over the parameter domain to find the optimal concrete value for the variable parameter. In Figure 4, we present an example to illustrate the parameter searching procedure. The cutting line in the figure represents the parameter in the constraint, which decides the ratio of tuples in the shadow area, i.e., satisfying the constraint. By increasing or decreasing the parameter, the likelihood of tuples in the shadow area changes correspondingly. The technical challenge behind the search is the hardness of likelihood evaluation over the satisfying tuples, or equivalently the probability of tuples falling in the shadow area in our example. To tackle the problem, we adopt the random sampling algorithm, which is also suited for the non-uniform distribution of the involved columns. Note that the binary search may not be able to find a parameter with the desired precision, based on the determined data distribution of columns after processing equality constraints over filters. Therefore, in Algorithm 1, we try to instantiate the parameters for non-equality constraints upon different data distributions by iteration.

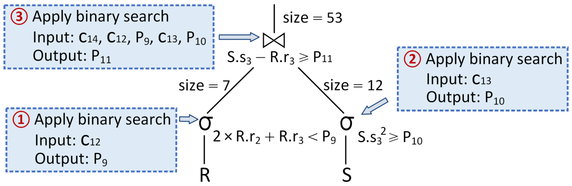


Figure 5: An example of parameter instantiation for non-equality constraints on join operator

The complexity of the approach is the product of two components, the number of iterations in parameter value search and the computational cost of probability evaluation using random sampling algorithm in each iteration. The number of iterations for the binary search is logarithmic to the domain size of the parameter, decided by the minimal and maximal value that the expression with multiple columns could reach. The cost of random sampling depends on the complexity of the predicate, which usually only involves a few columns.

Joins with Non-Equality Constraint are slightly different from the filters with non-equality constraints, because the columns involved in their predicates may overlap with the columns in the predicates of their child nodes as query Q_4 in Figure 1, which usually does not happen to filters in the query execution tree. Therefore, we must process the constraints in a bottom-up manner without the premise of probability independence, such that the precedent operators are settled before the join operator with non-equality constraint is handled. In Figure 5, we present the processing flow on query Q_4. After *Touchstone* concretizes the parameters P_9 and P_{10} in constraints c_{12} and c_{13}, the input data to the join operator with constraint c_{14} is determined. Based on the characteristics of the inputs, we apply the same binary search strategy designed for filter operator to construct the optimal parameter, e.g., P_{11} in Figure 5, for the desired result cardinality. Since the algorithm is identical to that for filter operator, we hereby skip detailed algorithm descriptions as well as the complexity analysis.

4 Data Generation

Given the generators of all non-key columns and the instantiated queries, the data generation component is responsible for assembling tuples based on the outputs of the column generators. The key technical challenge here is to meet the equality constraints over the join operators, i.e., $C^{\bowtie}_{=}$, which involve the dependencies among primary and foreign keys from multiple tables. To tackle the problem, we design a new tuple generation schema, which focuses on the manipulation of foreign keys only.

The tuple generation consists of two steps. In the first *compilation* step, *Touchstone* orders the tables as a generation sequence and decomposes the query trees into constraint chains for each target table. In the second *assembling* step, the working threads in *Touchstone* independently generate tuples for the tables based on the result order from compilation step. For each tuple, the working thread fills values in the columns by calling the random column generators independently and incrementally assigns a primary key, while leaving the foreign keys blank. By iterating the constraint chains associated with the table, the algorithm identifies the appropriate candidate keys for each foreign key based on the maintained join information of the referenced primary key, and randomly assigns one of the candidate keys to the tuple.

Compilation Step: The generation order of the tables is supposed to be consistent with the dependencies between primary keys and foreign keys, because the primary key must be generated before the adoption of its join information for generating corresponding foreign keys of other tables. Since such primary-foreign-key dependencies form a directed acyclic graph (DAG), *Touchstone* easily constructs a topological order over the tables. In Figure 6, we illustrate the result order over three tables, $R \to S \to T$, based on the database schema H in Figure 1.

In order to decouple the dependencies among columns and facilitate parallelizing, *Touchstone* decomposes the query trees annotated with constraints into *constraint chains*. A constraint chain consists of a number of constraints corresponding to the cardinality constraints over the operators in query trees. There are three types of constraints included in the constraint chains, namely FILTER, PK and FK, which are associated with the types of related operators. The constraint chains with respect to a table are defined as the sequences of constraints with descendant relationship in the query trees. In Figure 6, we present all the constraint chains for tables R, S and T. For example, table R has two constraint chains extracted from queries Q_1 and Q_3. And the constraint chains of table S are marked in Figure 1 for easily understanding.

Each FILTER constraint keeps the predicate with the instantiated parameters. Each PK constraint in the chain records the column name of the primary key. Each FK constraint maintains a triplet, covering two column names of the foreign key and the referenced primary key, and the expected *ratio* of tuples satisfying the predicate on the join operator. The second constraint in the first chain for table S in Figure 6, for example, is $\mathrm{FK}[S.s_2, R.r_1, \frac{2}{3}]$, indicating the foreign key is $S.s_2$, the referenced primary key is $R.r_1$ and two out of three tuples in table S are expected to meet the predicate $S.s_2 = R.r_1$ of join operator in the case of satisfying the predicate $S.s_3 = P_2$ of previous filter. The expected ratios in FK constraints are calculated based on the cardinality requirements of the specified cardinality constraints.

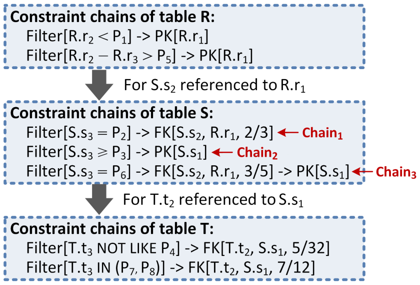

Constraint chains of table R:
Filter[R.r$_2$ < P$_1$] -> PK[R.r$_1$]
Filter[R.r$_2$ – R.r$_3$ > P$_5$] -> PK[R.r$_1$]
For S.s$_2$ referenced to R.r$_1$
Constraint chains of table S:
Filter[S.s$_3$ = P$_2$] -> FK[S.s$_2$, R.r$_1$, 2/3]
Chain$_1$
Filter[S.s$_3$ ⩾ P$_3$] -> PK[S.s$_1$]
Chain$_2$
Filter[S.s$_3$ = P$_6$] -> FK[S.s$_2$, R.r$_1$, 3/5] -> PK[S.s$_1$]
Chain$_3$
For T.t$_2$ referenced to S.s$_1$
Constraint chains of table T:
Filter[T.t$_3$ NOT LIKE P$_4$] -> FK[T.t$_2$, S.s$_1$, 5/32]
Filter[T.t$_3$ IN (P$_7$, P$_8$)] -> FK[T.t$_2$, S.s$_1$, 7/12]

Figure 6: Results of constraint chain decomposition

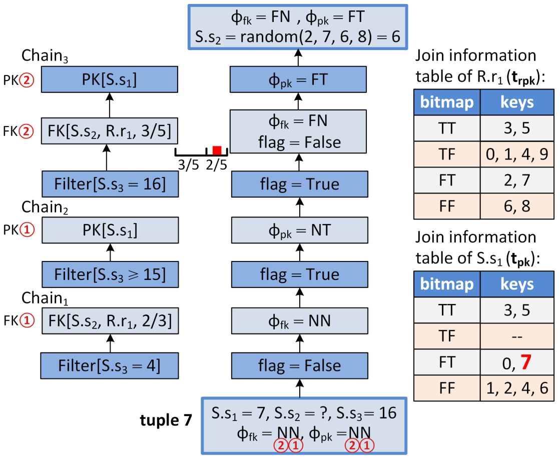
Assembling Step: For simplicity, we assume that there is a single-column primary key and one foreign key in the table. Note that our algorithm can be naturally extended to handle tables with composite primary key and multiple foreign keys. The result constraint chains are distributed to all working threads on multiple nodes for parallel tuple generation. When generating tuples for a specified table, each working thread maintains two bitmap data structures at runtime, i.e., ϕ_{fk} and ϕ_{pk}. They are used to keep track of the status of joinability, e.g., whether the generating tuple satisfies individual predicates over join operators, for primary key and foreign key, respectively. The length of the bitmap ϕ_{fk} (resp. ϕ_{pk}) is equivalent to the number of FK (resp. PK) constraints in all chains of the target table. Each bit in the bitmap corresponds to a FK/PK constraint. It has three possible values, T, F and N, indicating if the join status is successful, unsuccessful or null. In Figure 6, for example, table S has two FK constraints and two PK constraints, resulting in 2-bit representations for both ϕ_{fk} and ϕ_{pk}.

Touchstone also maintains the join information table to track the status of joinability of primary keys based on the bitmap representation ϕ_{pk}. In Figure 7, we show two join information tables of primary keys $R.r_1$ and $S.s_1$ respectively. The join information table of $R.r_1$ is maintained in the generation of table R, which is ready for generating the foreign key $S.s_2$ of table S. During the generation of table S, the join information table of $S.s_1$ is maintained for generating the foreign key $T.t_2$ of table T. There are two attributes in the entry of join information table, i.e., bitmap and keys, indicating the status of joinability and the corresponding satisfying primary key values. Note that the keys in the entry may be empty (such entries will not be stored in practice), which means there is no primary key with the desired joinability status.

Algorithm 2 Tuple generation

```
Input: Column generators $\bar{G}$, constraint chains of the target table $\Omega$, join information tables of referenced primary key and current primary key $t_{rpk}$ and $t_{pk}$
Output: Tuple $r$ and join information table $t_{pk}$
1:  $r.pk \leftarrow$ a value assigned incrementally
2:  $r.columns \leftarrow$ values output by column generators $\bar{G}$
3:  $\phi_{fk} \leftarrow N...N$, $\phi_{pk} \leftarrow N...N$
4:  for all constraint chain $\omega \in \Omega$ do
5:      $flag \leftarrow True$
6:      for all constraint $c \in \omega$ do
7:          if ($c$ is FILTER) && ($c.predicate$ is $False$) then
8:              $flag \leftarrow False$
9:          else if $c$ is PK then
10:             $\phi_{pk}[i] \leftarrow flag$  // $i$ is the bit index for $c$
11:         else if ($c$ is FK) && $flag$ then
12:             if $random[0,1) \geq c.ratio$ then $flag \leftarrow False$
13:             $\phi_{fk}[i] \leftarrow flag$  // $i$ is the bit index for $c$
14: $r.fk \leftarrow$ a value selected from $t_{rpk}$ satisfying $\phi_{fk}$
15: Add $r.pk$ in the entry of $t_{pk}$ with bitmap $\phi_{pk}$
16: return $r$ and $t_{pk}$
```

The tuple generation algorithm is listed in Algorithm 2. We present a running example of tuple generation in Figure 7. A new tuple for table S is initialized as $(S.s_1 = 7, S.s_2 = ?, S.s_3 = 16)$, $\phi_{fk} = NN$ and $\phi_{pk} = NN$ (lines 1-3). The $flag$ is set to True before traversing each constraint chain (line 5), which is used to track if the predicates from the precedent constraints of current chain are fully met. On the first constraint chain, since the predicate in the first FILTER constraint is $S.s_3 = 4$, $flag$ is then set to False (line 8), and algorithm does not need to handle the next FK constraint (line 11). On the second chain, the tuple satisfies the predicate $S.s_3 \geq 15$, resulting in the update of bitmap representation as $\phi_{pk} = NT$ (line 10). On the third chain, after passing the first FILTER constraint, the corresponding bit of next FK constraint in ϕ_{fk} is randomly flipped to F at the probability of $\frac{2}{5}$ (lines 12-13), because the expected $ratio$ of satisfying tuples is $\frac{3}{5}$. The $flag$ is set to False (line 12) to reflect the failure of full matching of precedent constraints for later PK constraint. Then, the bit corresponding to next PK constraint in ϕ_{pk} is set as F according to the value of $flag$ (line 10). Therefore, the two bitmaps are finalized as $\phi_{fk} = FN$ and $\phi_{pk} = FT$. Then the algorithm identifies (line 14) two entries matching $\phi_{fk} = FN$, namely satisfying the T/F requirements on the corresponding bits of ϕ_{fk}, with bitmaps FT and FF respectively, in the join information table of $R.r_1$. Given these two entries, it randomly selects (line 14) a foreign key, e.g., 6, from four candidate referenced primary keys $\{2,7,6,8\}$, which are all appropriate as the foreign key $S.s_2$. That there is no entry in t_{rpk} satisfying the T/F requirements of ϕ_{fk}, which is called *mismatch case*, is dealt in the rest of the section. Finally, the algorithm updates (line 15) the join information table of $S.s_1$ by adding the primary key $S.s_1 = 7$ into the entry with bitmap FT.



Join information table of R.r1 (**t**rpk):

bitmap	keys
TT	3, 5
TF	0, 1, 4, 9
FT	2, 7
FF	6, 8

Join information table of S.s1 (**t**pk):

bitmap	keys
TT	3, 5
TF	--
FT	0, **7**
FF	1, 2, 4, 6

Figure 7: Running example of tuple generation for table S

For a table, suppose there are k non-key columns, m constraints in the related constraint chains and n entries in the join information table of referenced primary key. The complexity of tuple generation mainly consists of three parts, k times of calling random column generators for filling the values of non-key columns, the traversing over m constraints within chains for determining the joinability statuses of foreign key and primary key, and the comparing with n bitmaps in the join information table for searching the appropriate foreign key candidates. For practical workloads, k, m and n are all small numbers, e.g., $k \leq 12$, $m \leq 20$ and $n \leq 40$ for TPC-H [3] workload, so our tuple generation is highly efficient.

Handling Mismatch Cases: For the data generation of big tables, if a joinability status of the primary key may occur, its occurrence can be considered as inevitable based on the probability theory. However, there are still some joinability statuses of the primary key that never occur. For example, in Figure 7, the bitmap ϕ_{pk} for primary key $S.s_1$ can not be TF due to the constraints, i.e., Filter[$S.s_3 = 16$] and Filter[$S.s_3 \geq 15$]. Therefore, in the tuple generation, it should be avoided to generate the bitmap ϕ_{fk} that does not have any matching entry in the join information table of the referenced primary key. In order to achieve this objective, the main idea is to add rules to manipulate relevant FK constraints.

Figure 8 gives an example of adjustments to FK constraints for handling the mismatch case. There are three FK constraints with the serial numbers of 1, 2 and 3 in the three constraint chains, respectively. Since there are four bitmaps, i.e., FTT, TTT, TFT, FTF, that are not presented in the join information table of the referenced primary key rpk corresponding to the foreign key fk of the target table, three rules are added in two FK constraints to avoid

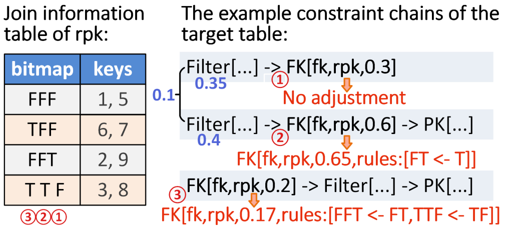

Join information table of rpk:
The example constraint chains of the target table:
bitmap
keys
FFF
1, 5
TFF
6, 7
FFT
2, 9
T T F
3, 8
③②①
Filter[...] -> FK[fk,rpk,0.3]
0.35
①
No adjustment
0.1
Filter[...] -> FK[fk,rpk,0.6] -> PK[...]
0.4
②
FK[fk,rpk,0.65,rules:[FT <- T]]
FK[fk,rpk,0.2] -> Filter[...] -> PK[...]
③
FK[fk,rpk,0.17,rules:[FFT <- FT,TTF <- TF]]

Figure 8: An example of adjustments to FK constraints

producing any ϕ_{fk} triggering the mismatch case. For example, there is a rule $[FT \leftarrow T]$ added in the second FK constraint, which indicates that the status of the second FK constraint must be F if the status of the first FK constraint is T in the tuple generation. Since there are extra F statuses forcibly generated by the added rule for the second FK constraint, the actual ratio of tuples satisfying the corresponding predicate could be lower than the expected ratio 0.6. Consequently, it is necessary to adjust the ratio in the second FK constraint for eliminating the impact of the added rule. In this example, we adjust the ratio as $0.65 = \frac{0.6 \times 0.4}{0.4 - 0.1 \times 0.3}$, in which 0.4 is the ratio of tuples satisfying the predicate in the second FILTER constraint, 0.6×0.4 is the cumulative probability of the status T for the second FK constraint, 0.1 is the ratio of tuples satisfying the two predicates in the first two FILTER constraints, 0.3 is the ratio in the first FK constraint and 0.1×0.3 is the cumulative probability of the extra F status generated by the rule. The general algorithm of adjustments to FK constraints and the corresponding analyses are presented in our online technical report [2].

To reflect the adjustments to FK constraints in the tuple generation, minor modification is applied on the original tuple generation algorithm on lines 12-13 in Algorithm 2. Specifically, the updated algorithm first checks all existing rules in current FK constraint. If there is a rule which can be applied to the statuses of previous constraints, ϕ_{fk} and $flag$ are updated according to the rule. Otherwise, the algorithm updates ϕ_{fk} and $flag$ by the probability based on the adjusted ratio.

Management of Join Information: For generation of a table, it can be completely parallel on multiple nodes with multiple working threads on each node. Each working thread maintains its own join information table of the primary key to avoid contention. But the join information table of referenced primary key can be shared among multiple working threads on each node. After the generation of the table, we merge the join information tables maintained by the multiple working threads in distributed controller as in Figure 2. But there are serious memory and network problems for the space complexity of the join information table is $O(s)$ with s as the table size.

Since the relationship of foreign key and primary key can be many to one and the intermediate result cardinality is the main factor that affects the query performance, we design a compression method by storing less primary key values in the join information table but still promise the randomness of remaining values. Assuming the size of keys in an entry of join information table is N, which is hard to know in advance and may be very large. We aim to store only L ($L << N$) values in the keys and promise the approximately uniform distribution of these L ones among all N values. The compression method is implemented as follows: we store the first L arriving values in the keys, if any; and for the i-th ($i > L$) arriving value, we randomly replace a value stored previously in the keys with the probability of L/i. By such a method, the space complexity of the join information table is reduced to $O(n * L)$, where n is the number of entries in the join information table and L is the maximum allowed size of keys in each entry. Since n is generally small, e.g., $n \leq 40$ for TPC-H workload, and L usually can be set to thousands, the memory consumption and network transmission of the join information table are acceptable.

5 Experiments

Environment. Our experiments are conducted on a cluster with 8 nodes. Each node is equipped with 2 Intel Xeon E5-2620 @ 2.0 GHz CPUs, 64GB memory and 3 TB HDD disk configured in RAID-5. The cluster is connected using 1 Gigabit Ethernet.

Workloads. The TPC-H [3] is a decision support benchmark which contains the most representative queries of analytical applications, while the transactional benchmarks, e.g., TPC-C and TPC-W, do not contain queries for analytical processing. So we take the TPC-H workload for our experiments. We compare *Touchstone* with the state-of-the-art work MyBenchmark [15] with source codes from the authors.[2] The workloads for comparison consist of 6 queries from TPC-H, including $Q_{2,3,6,10,14,16}$. Note that these queries are selected based on the performance of MyBenchmark, which drops significantly when other queries are included in the workloads. *Touchstone*, on the other hand, can easily handle all of the first 16 queries, i.e., Q_1 to Q_{16}, in TPC-H with excellent performance. To the best of our knowledge, *Touchstone* provides the widest support to TPC-H workload, among all the existing studies [6, 14, 5, 15].

Input Generation. To build valid inputs for experiments, we generate the DBI and queries of TPC-H using its tools *dbgen* and *qgen*, respectively. And the DBI of TPC-H is imported into the MySQL database. The database schema of TPC-H is used as the input H. We can easily obtain the input data characteristics D for all columns from the DBI in MySQL. Given the TPC-H queries, their physical query plans are obtained from MySQL query parser and optimizer over the DBI. The

[2] We would like to thank Eric Lo for providing us the source code of MyBenchmark.

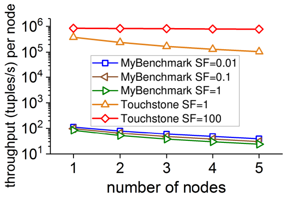


Figure 9: Comparison of data generation throughput

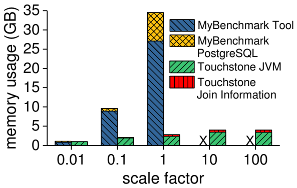


Figure 10: Comparison of memory consumption

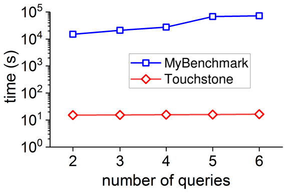


Figure 11: Comparison of data generation time

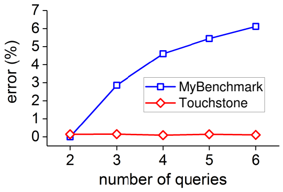


Figure 12: Comparison of global relative error

cardinality constraints corresponding to the operators in query plans are then identified by running the queries on the DBI in MySQL. The input workload characteristics W are constructed by the parameterized TPC-H queries and above cardinality constraints. Note that we can generate databases with different scale factors using the same input W by employing selectivities instead of the absolute cardinalities in our input constraints.

Settings. As data is randomly generated according to the column generators in *Touchstone*, the distribution of generated data may be difficult to satisfy the expectation for small tables such as *Region* and *Nation*. We therefore revise the sizes of *Region* and *Nation* from 5 to 500, and from 25 to 2500 respectively. The cardinality constraints involving these two tables are updated proportionally. In addition, the small tables can also be pre-generated manually. The error threshold (desired precision) and maximal iterations in query instantiation are set to 10^{-4} and 20 respectively. The default maximum allowed size L of keys in join information table is set to 10^4.

5.1 Comparison with MyBenchmark

We compare *Touchstone* with MyBenchmark from four aspects, including data generation throughput, scalability to multiple nodes, memory consumption and capability of complex workloads.

Figure 9 shows the data generation throughputs per node of *Touchstone* and MyBenchmark as we vary the number of nodes under different scale factors. Due to the unacceptably long processing time of MyBenchmark, we adopt smaller scale factors for it and large scale factors for *Touchstone*. Overall, the data generation throughput of *Touchstone* is at least 3 orders of magnitude higher than that of MyBenchmark. This is because MyBenchmark does not have a good parallelization or an efficient data generation schema. Furthermore, as the number of nodes increases from 1 to 5, the data generation throughput per node of MyBenchmark decreases dramatically for all three scale factors. Although the decline of data generation throughput per node of *Touchstone* is obvious too when $SF = 1$, *Touchstone* is linearly scalable (the throughput per node is stable) when $SF = 100$. This is because for small target database, e.g., $SF = 1$, the distributed maintenance rather than data generation dominates the computational cost in *Touchstone*, while its overhead comparatively diminishes by increasing the target database size.

Figure 10 reports the peak memory consumptions of *Touchstone* and MyBenchmark under different data scales. The experiment is conducted on 5 nodes with no restriction on memory usage. The memory usage of MyBenchmark mainly consists of two parts, namely, memory consumed by MyBenchmark Tool and memory consumed by PostgreSQL for managing intermediate states. The memory usage of *Touchstone* mainly includes memory for JVM itself and memory for maintaining join information. As shown in Figure 10, the memory consumption of *Touchstone* is much lower than that of MyBenchmark under the same scale factors. It is worth noting that the memory consumption of *Touchstone* remains almost constant when $SF > 10$. This is because for *Touchstone*, the JVM itself occupies most of the memory, while the join information maintenance only spends a tiny piece of memory.

Figure 11 and Figure 12 present the data generation time (total running time) and global relative error separately of *Touchstone* and MyBenchmark as we vary the number of input queries with $SF = 1$. The input queries are loaded in order of their serial numbers. The experiment is carried out on 5 nodes. In Figure 11, it is obvious that the data generation time of MyBenchmark increases significantly as the number of queries increases. At the same time, the generation time of *Touchstone* grows very little when more queries are included, significantly outperforming MyBenchmark. In Figure 12, the error of *Touchstone* is much smaller than that of MyBenchmark. Moreover, as there are more input queries, the global relative error of *Touchstone* remains small with little change, while the error of MyBenchmark has an obvious rise. In summary, *Touchstone* is more capable of supporting complex workloads than MyBenchmark.

It can be seen from previous experiments that MyBenchmark can not be easily applied to generate the terabyte scale database for complex workloads due to its poor performance. In the following, we further demonstrate the advantages of *Touchstone* by a series of experiments using the workload of 16 queries, i.e., Q_1 to Q_{16}.

5.2 Performance Evaluation

In this section, we evaluate the impact of workload complexity on query instantiation time and total running time

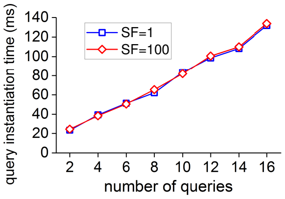


Figure 13: Query instantiation time

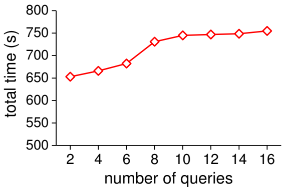


Figure 14: Total running time

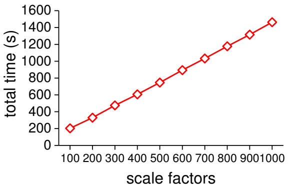


Figure 15: Scalability to data scale

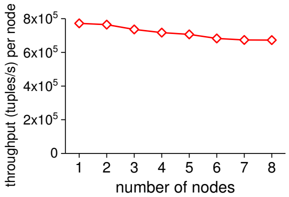


Figure 16: Scalability to multiple nodes

in *Touchstone*, as well as the scalability to data scale and multiple nodes of *Touchstone*.

Figure 13 shows the query instantiation time of *Touchstone* as we vary the number of queries with $SF = 1$ and $SF = 100$, respectively. The input queries are loaded in order of their serial numbers. The query instantiator is deployed on a single node. As shown in Figure 13, even when all 16 queries are used for input, query instantiation is finished within 0.2s. And there is a minimal difference in query instantiation time for $SF = 1$ and $SF = 100$, as the complexity of query instantiation is independent of data scale. Overall, the query instantiation time is only correlated to the complexity of input workloads.

Figure 14 shows the total running time of *Touchstone* as we vary the number of queries with $SF = 500$. *Touchstone* is deployed on 8 nodes. From the result, it can be seen that the running time increases slowly as the number of queries increases. For Q_7 and Q_8, there are relatively more cardinality constraints over equi-join operators, so the time increment is larger when we change from 6 queries to 8 queries. But when the number of queries changed from 10 to 16, the time increment is almost indiscernible, for Q_{11} to Q_{16} are simple, among which Q_{12} to Q_{15} have no cardinality constraints on equi-join operators[3]. Overall, the total running time increased by only 16% from 2 queries to 16 queries for 500GB data generation task, so *Touchstone* is insensitive to the workload complexity.

Figure 15 presents the total running time of *Touchstone* under different scale factors with the input of 16 queries. *Touchstone* is deployed on 8 nodes. As shown in Figure 15, *Touchstone* is linearly scalable to data size. Because the generation of each tuple is independent and the generated tuples need not be stored in memory, the data generation throughput is stable for different data scales. Moreover, the total runtime of *Touchstone* is less than 25 minutes for $SF = 1000$ (1TB), so it is capable of supporting industrial scale database generation.

Figure 16 presents the data generation throughputs per node of *Touchstone* as we vary the number of nodes with $SF = 500$. The input workload includes 16 queries. The result shows that the data generation throughput per node is approximatively unchanged as the number of nodes increases, validating the linear scalability of *Touchstone*. To the best of our knowledge, *Touchstone* is the first query-aware data generator which can support full parallel data generation on multiple nodes.

5.3 Data Fidelity Evaluation

The data fidelity of synthetic database is evaluated by relative error on cardinality constraints and performance deviation on query latencies. We calculate the relative error for each query in the similar way with global relative error, which only involves its own cardinality constraints. We compare the latency of query processing on base database generated by *dbgen* against that on synthetic database generated by *Touchstone* to show the performance deviation.

Figure 17 shows the relative errors for Q_1 to Q_{16} with different scale factors from 1 to 5. The maximum error among all 16 queries is less than 4%, and there are 14 queries with errors less than 1%. Figure 18 shows the global relative error of all 16 queries as we vary the scale factor, which is less than 0.2% for all scale factors. And with the increase of scale factor, the global relative error has a sharp decrease. Since data is randomly generated by column generators, as expected by the probability theory, the larger the data size, the smaller the relative error.

Figure 19 presents the performance deviations of all 16 queries with $SF = 1$. We vary the maximum allowed size L of keys in the join information table from 10^3 to 10^5. We can see that the performance deviation is inconspicuous for all 16 queries, and the size of L has no significant influence on query latencies. The result strongly illustrates the correctness and usefulness of our work. We are the first work to give such an experiment to verify the fidelity of the generated DBI.

More experimental results are available in our online technical report [2], which demonstrate the effectiveness for data generation of non-equi-join workloads, handling mismatch cases, the compression method on join information table, and other benchmark workloads.

6 Related Work

There are many data generators [7, 12, 11, 4, 20, 23, 1, 9] which only consider the data characteristics of the target database. For example, Alexander et al. [4] proposes pseudo-random number generators to realize the parallel

[3] Depending on the physical query plans of Q_{12} to Q_{15}, the primary keys in their equi-join operators are from the original tables, so all foreign keys must be joined and the sizes of output tuples are determined.

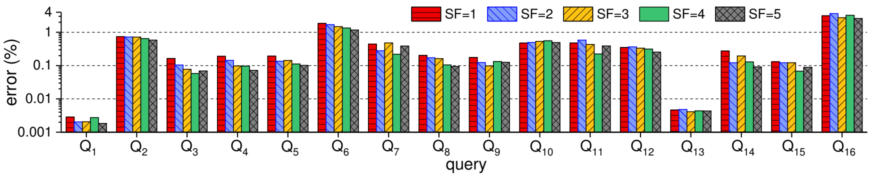


Figure 17: Relative error for each query

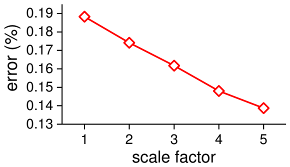


Figure 18: Global relative error vs. scale factor

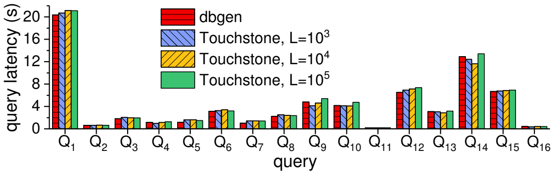


Figure 19: Performance deviation for each query

data generation. Torlak [23] supports the scalable generation of test data from a rich class of multidimensional models. However, all these data generators can not generate test databases with the specified workload characteristics on target queries.

There are query-aware data generators [6, 14, 5, 15], among which [6, 14, 15] are a series of work. QAGen [6] is the first query-aware data generator, but for each query it generates an individual DBI and its CSP (constraint satisfaction program) has the usability limitations as declared in experimental results. WAGen [14] makes a great improvement that it generates m ($\leq n$) DBIs with n input queries, but WAGen can't guarantee that only one DBI is generated and still has CSP performance problem. Though MyBenchmark [15] has done a lot of performance optimization, generating one DBI can not be promised for multiple queries and the performance is still unacceptable for the generation of terabyte scale database. DCGen [5] uses a novel method to represent data distribution with ideas from the probabilistic graphical model. But DCGen is weak in support of foreign key constraint, and it cannot easily support parallel data generation in a distributed environment.

There are some interesting non-relational data generators [18, 8, 13, 19, 10]. For example, Olston et al. [18] introduces how to generate example data for dataflow programs. Sara [8] generates structural XML documents. [13, 19] are synthetic graph generators. Chronos [10] can generate stream data for real time applications. In addition, there are query generation works [17, 16] which are partly similar to us, but they generate queries satisfying the specified cardinality constraints over an existing DBI. Moreover, the dataset scaling works [22, 25] can serve part of our targets, which scale up/down a given DBI with similar column correlations.

7 Discussion and Conclusion

Limitations. *Touchstone* aims to support the most common workloads in real world applications. Below we list the scenarios that we cannot support currently. (1) *Touchstone* does not support filters on key columns. Primary and foreign keys are identifiers of tuples and generally have no physical meaning, so the filters which are representations of business logics usually do not involve key columns. (2) Equality constraints over filters involving multiple columns are not supported in *Touchstone*. The equality predicate with multiple columns for filter is a very strict constraint, and has not been found in workloads of standard benchmarks. (3) Equi-joins on columns with no reference constraint are not supported in our work. This is because the equi-join is usually applied on the pair of primary and foreign keys in practical workloads, which is also the assumption of many works [5, 24, 25]. (4) *Touchstone* does not support the database schema with cyclic reference relationship. In our data generation process, generating foreign keys requires the join information tables of corresponding referenced primary keys, so the primary-foreign-key dependencies must form a direct acyclic graph (DAG), which is also the precondition of DCGen [5].

Privacy issue. Our work can help to protect privacy to some extend by removing query parameter values or using approximate query intermediate cardinalities. However, if the database statistics and workload characteristics are strictly related to privacy issues in some cases, it will not be a good way to use this kind of workload-aware data generators for performance testing.

In this paper we introduce *Touchstone* [2], a query-aware data generator with characteristics of completely parallelizable and bounded usage to memory. And *Touchstone* is linearly scalable to computing resource and data scale. Our future work is to support more operators, e.g., intersect and having, for covering the complex queries of TPC-DS, which has not be well supported by any existing query-aware data generation work.

Acknowledgements: Rong Zhang and Aoying Zhou are supported by National Key Research and Development Plan Project (No. 2018YFB1003303), and National Science Foundation of China (NSFC) (No.61332006, 61672233 and 61732014). And this work was done when Zhenjie Zhang and Xiaoyan Yang were with Advanced Digital Sciences Center, supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

References

[1] DTM data generator. http://www.sqledit.com/dg/.

[2] Technical report, running examples and source code of Touchstone. https://github.com/daseECNU/Touchstone.

[3] TPC-H benchmark. http://www.tpc.org/tpch/.

[4] ALEXANDROV, A., TZOUMAS, K., AND MARKL, V. Myriad: scalable and expressive data generation. *Proceedings of the VLDB Endowment 5*, 12 (2012), 1890–1893.

[5] ARASU, A., KAUSHIK, R., AND LI, J. Data generation using declarative constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (2011), ACM, pp. 685–696.

[6] BINNIG, C., KOSSMANN, D., LO, E., AND ÖZSU, M. T. Qagen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (2007), ACM, pp. 341–352.

[7] BRUNO, N., AND CHAUDHURI, S. Flexible database generators. In *Proceedings of the 31st international conference on Very large data bases* (2005), VLDB Endowment, pp. 1097–1107.

[8] COHEN, S. Generating xml structure using examples and constraints. *Proceedings of the VLDB Endowment 1*, 1 (2008), 490–501.

[9] GRAY, J., SUNDARESAN, P., ENGLERT, S., BACLAWSKI, K., AND WEINBERGER, P. J. Quickly generating billion-record synthetic databases. In *ACM SIGMOD Record* (1994), vol. 23, ACM, pp. 243–252.

[10] GU, L., ZHOU, M., ZHANG, Z., SHAN, M.-C., ZHOU, A., AND WINSLETT, M. Chronos: An elastic parallel framework for stream benchmark generation and simulation. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on* (2015), IEEE, pp. 101–112.

[11] HOAG, J. E., AND THOMPSON, C. W. A parallel general-purpose synthetic data generator. *ACM SIGMOD Record 36*, 1 (2007), 19–24.

[12] HOUKJÆR, K., TORP, K., AND WIND, R. Simple and realistic data generation. In *Proceedings of the 32nd international conference on Very large data bases* (2006), VLDB Endowment, pp. 1243–1246.

[13] LESKOVEC, J., CHAKRABARTI, D., KLEINBERG, J., AND REALISTIC, C. F. Mathematically tractable graph generation and evolution, using kronecker multiplication european conf. on principles and practice of know. dis. *Databases (ECML/PKDD)* (2005).

[14] LO, E., CHENG, N., AND HON, W.-K. Generating databases for query workloads. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 848–859.

[15] LO, E., CHENG, N., LIN, W. W., HON, W.-K., AND CHOI, B. Mybenchmark: generating databases for query workloads. *The VLDB Journal 23*, 6 (2014), 895–913.

[16] MISHRA, C., AND KOUDAS, N. Interactive query refinement. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (2009), ACM, pp. 862–873.

[17] MISHRA, C., KOUDAS, N., AND ZUZARTE, C. Generating targeted queries for database testing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 499–510.

[18] OLSTON, C., CHOPRA, S., AND SRIVASTAVA, U. Generating example data for dataflow programs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (2009), ACM, pp. 245–256.

[19] PHAM, M.-D., BONCZ, P., AND ERLING, O. S3g2: A scalable structure-correlated social graph generator. In *Technology Conference on Performance Evaluation and Benchmarking* (2012), Springer, pp. 156–172.

[20] SHEN, E., AND ANTOVA, L. Reversing statistics for scalable test databases generation. In *Proceedings of the Sixth International Workshop on Testing Database Systems* (2013), ACM, p. 7.

[21] SYRJÄNEN, T. Logic programs and cardinality constraints–theory and practice.

[22] TAY, Y., DAI, B. T., WANG, D. T., SUN, E. Y., LIN, Y., AND LIN, Y. Upsizer: Synthetically scaling an empirical relational database. *Information Systems 38*, 8 (2013), 1168–1183.

[23] TORLAK, E. Scalable test data generation from multidimensional models. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (2012), ACM, p. 36.

[24] ZAMANIAN, E., BINNIG, C., AND SALAMA, A. Locality-aware partitioning in parallel database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 17–30.

[25] ZHANG, J., AND TAY, Y. Dscaler: Synthetically scaling a given relational database. *Proceedings of the VLDB Endowment 9*, 14 (2016), 1671–1682.

DSAC: Effective Static Analysis of Sleep-in-Atomic-Context Bugs in Kernel Modules

Jia-Ju Bai[1], Yu-Ping Wang[1], Julia Lawall[2], Shi-Min Hu[1]
[1]Tsinghua University, [2]Sorbonne Université/Inria/LIP6

Abstract

In a modern OS, kernel modules often use spinlocks and interrupt handlers to monopolize a CPU core to execute concurrent code in atomic context. In this situation, if the kernel module performs an operation that can sleep at runtime, a system hang may occur. We refer to this kind of concurrency bug as a sleep-in-atomic-context (SAC) bug. In practice, SAC bugs have received insufficient attention and are hard to find, as they do not always cause problems in real executions.

In this paper, we propose a practical static approach named DSAC, to effectively detect SAC bugs and automatically recommend patches to help fix them. DSAC uses four key techniques: (1) a hybrid of flow-sensitive and -insensitive analysis to perform accurate and efficient code analysis; (2) a heuristics-based method to accurately extract kernel interfaces that can sleep at runtime; (3) a path-check method to effectively filter out repeated reports and false bugs; (4) a pattern-based method to automatically generate recommended patches to help fix the bugs.

We evaluate DSAC on kernel modules (drivers, file systems, and network modules) of the Linux kernel, and on the FreeBSD and NetBSD kernels, and in total find 401 new real bugs. 272 of these bugs have been confirmed by the relevant kernel maintainers, and 43 patches generated by DSAC have been applied by kernel maintainers.

1. Introduction

Concurrency bugs are known to be difficult to debug. Many tools have been proposed to detect common concurrency bugs such as atomicity violations and data races. However, as a kind of concurrency bug, sleep-in-atomic-context (SAC) bugs have received less attention. SAC bugs occur at the kernel level when a sleeping operation is performed in atomic context [10], such as when holding a spinlock or executing an interrupt handler. Code executing in atomic context monopolizes a CPU core, and the progress of other threads that need to concurrently access the same resources is delayed. Thus the code execution in atomic context should complete as quickly as possible. Sleeping in atomic context is forbidden, as it can block a CPU core for a long period and may lead to a system hang.

Even though sleeping in atomic context is forbidden, many SAC bugs still exist, especially in kernel modules, such as device drivers and file systems. The main reasons why SAC bugs continue to occur include: (1) Determining whether an operation can sleep often requires system-specific experience; (2) Testing kernel modules can be difficult, for example, running a device driver requires its associated device; (3) SAC bugs do not always cause problems in real execution, and they are often hard to reproduce at runtime. Recent studies [12, 48] have shown that SAC bugs have caused serious system hangs at runtime. Thus, it is necessary to detect and fix SAC bugs in kernel modules.

Many existing approaches [7, 19, 28, 42] can detect concurrency bugs, but most of them are designed for user-level applications. Some approaches [13, 17, 18, 41, 44] can detect some common kinds of kernel-level concurrency bugs, such as atomicity violations and data races, but they have not addressed SAC bugs. Several approaches [2, 9, 16, 34, 53] can detect common kinds of OS kernel faults, including SAC bugs. But they are not specific to SAC bugs, and most of them [9, 16, 34] are designed to collect statistics rather than report specific bugs to the user, making issues such as detection time and false positive rate less important.

In this paper, we propose a static approach named DSAC[1] that targets accurately and efficiently detecting SAC (sleep-in-atomic-context) bugs in kernel modules, and can automatically recommend patches to help fix the detected bugs. DSAC consists of four phases. Firstly, DSAC uses a *hybrid of flow-sensitive and -insensitive analysis* (subsequently referred to as a *hybrid flow analysis*) to analyze the source code, in order to collect the set of functions that are possibly called in atomic context. Secondly, from the collected functions, DSAC exploits a *heuristics-based* method, which uses some heuristics based on the analysis of the call graphs and comments of the kernel code, to extract kernel interfaces that can sleep at runtime. Thirdly, with the extracted

[1] DSAC website: *http://oslab.cs.tsinghua.edu.cn/DSAC/index.html*

sleep-able kernel interfaces, DSAC first reuses the hybrid flow analysis to detect possible bugs, and then uses a *path-check* method to filter out repeated reports and false bugs by validating the code path of each detected bug. Finally, DSAC exploits a *pattern-based* method to automatically generate patches to help fix the bugs. This method analyzes the bug reports generated in the previous phase, and uses common fixing patterns to correct the buggy code.

We have implemented DSAC using LLVM [51]. To validate its effectiveness, we first evaluate DSAC on Linux drivers, which are typical of modules in the Linux kernel. To validate the generality and portability, we then use DSAC to check file systems and network modules in the Linux kernel, and finally use DSAC in FreeBSD and NetBSD to check their kernel source code. The results show that DSAC can indeed accurately and efficiently find real SAC bugs and recommend a number of correct patches to help fix the bugs.

DSAC has four main advantages in practical use:

1) Efficient and accurate code analysis. DSAC uses an efficient inter-procedural and context-sensitive analysis to maintain a lock stack across function calls, which can accurately identify the code in atomic context. All source files of the kernel module are analyzed at once to perform accurate analysis across function calls.

2) Precise and detailed bug reports. To achieve precise bug detection, DSAC uses a heuristics-based method to extract sleep-able kernel interfaces, and uses a path-check method to filter out repeated reports and false bugs. It also produces detailed reports of the found bugs, including code paths and source file names, for the user to locate and check.

3) Recommended patch generation. With the generated bug reports, DSAC uses a pattern-based method to automatically generate patches to help fix the detected bugs, which can reduce the manual work of bug fixing.

4) High automation, generality and portability. Once the user offers the names of spin-lock and -unlock functions, interrupt-handler-register functions and basic sleep-able kernel interfaces, the remaining phases of DSAC are fully automated. DSAC can effectively check kernel modules, including drivers, file systems and network modules. And it can also be easily ported in another OS to check the kernel code.

In this paper, we make three main contributions:

- We first analyze the challenges in detecting SAC bugs in kernel modules, and then propose four key techniques to address these challenges: (1) a hybrid flow analysis to perform accurate and efficient code analysis; (2) a heuristics-based method to accurately extract sleep-able kernel interfaces in the analyzed kernel modules; (3) a path-check method to effectively filter out repeated reports and false bugs; (4) a pattern-based method to automatically generate recommended patches to help fix the bugs.
- Based on the four techniques, we propose a practical approach named DSAC, to accurately and efficiently detect SAC bugs in kernel modules and automatically recommend patches to help fix the bugs.
- We evaluate DSAC on drivers in Linux 3.17.2 and 4.11.1. We select these kernel versions as they are near the beginning of stable series, and thus the simplest bugs should have been fixed in them. We find 200 and 320 real bugs respectively in these versions. 50 real bugs in 3.17.2 have been fixed in 4.11.1, and 209 real bugs in 4.11.1 have been confirmed by kernel maintainers. To validate the generality and portability, we use DSAC to check file systems and network modules in the Linux kernel, and then run it in FreeBSD 11.0 and NetBSD 7.1 to check their kernel code, and find 81 new real bugs. 43 generated patches for the three OS kernels have been applied by kernel maintainers.

The remainder of the paper is organized as follows. Section 2 presents the background. Section 3 presents the challenges and our techniques. Section 4 introduces DSAC in detail. Section 5 presents the evaluation. Section 6 compares DSAC to previous approaches. Section 7 presents limitations and future work. Section 8 gives the related work. Section 9 concludes this paper.

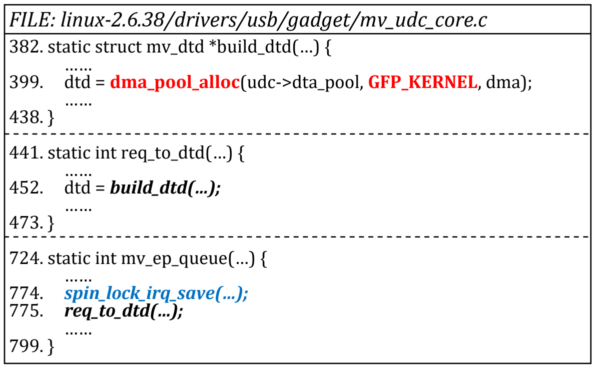
2. Background

In this section, we first introduce atomic context, and then motivate our work by an example of a real SAC bug in a Linux driver.

2.1 Atomic Context

Atomic context is an OS kernel state that a CPU core is monopolized to execute the code, and the progress of other threads that need to concurrently access the same resources is delayed. This context can protect resources from concurrent access, in which the code execution should complete as quickly as possible without able to be rescheduled. Due to this special situation, sleeping in atomic context is forbidden, as it can block CPU cores for long periods and may lead to a system hang.

There are two common examples of atomic context in the kernel, namely *holding a spinlock* and *executing an interrupt handler*. If a thread sleeps when holding a spinlock, another thread that requests the same spinlock will spin on a CPU core to wait until the former thread releases the spinlock. If threads spin on all CPU cores like this, no CPU core will be available for the former thread to release the spinlock, causing a deadlock [11]. If an interrupt handler sleeps, the kernel scheduler cannot reschedule it and a system hang may occur, as the interrupt handler is not backed by a process [29].



```
FILE: linux-2.6.38/drivers/usb/gadget/mv_udc_core.c
382. static struct mv_dtd *build_dtd(...) {
         ......
399.     dtd = dma_pool_alloc(udc->dta_pool, GFP_KERNEL, dma);
         ......
438. }
-----------------------------------------------------------
441. static int req_to_dtd(...) {
         ......
452.     dtd = build_dtd(...);
         ......
473. }
-----------------------------------------------------------
724. static int mv_ep_queue(...) {
         ......
774.     spin_lock_irq_save(...);
775.     req_to_dtd(...);
         ......
799. }
```

Figure 1: Part of the *usb_gadget* driver code in Linux 2.6.38.

Note that atomic context only occurs at the kernel level, as user-level applications are regularly interrupted by the OS scheduler when their time slices end. Though kernel developers often know that sleeping is not allowed in atomic context, many SAC bugs still exist [16, 34], especially in kernel modules.

2.2 Motivating Example

We motivate our work by a real bug in the *usb_gadget* that persisted over 8 releases (1.5 years) from Linux 2.6.38 to Linux 3.7. Figure 1 presents part of the source code for the driver. The function *mv_ep_queue* calls *spin_lock_irqsave* to take a spinlock (line 774) and then calls *req_to_dtd* (line 775). The function *req_to_dtd* calls *build_dtd* (line 452), which calls *dma_pool_alloc* with *GFP_KERNEL* to request a DMA memory pool (line 399). According to the kernel documentation [50], *dma_pool_alloc* called with *GFP_KERNEL* can sleep, thus a SAC bug exists. This bug was first fixed in Linux 3.7, by replacing *GFP_KERNEL* with *GFP_ATOMIC*, which indicates to *dma_pool_alloc* that it cannot sleep.

This example illustrates three main reasons why SAC bugs occur in kernel modules. (1) Determining whether an operation can sleep requires OS-specific knowledge. In this example, without experience in Linux kernel development, it may be hard to know that the function *dma_pool_alloc* called with *GFP_KERNEL* can sleep at runtime. (2) SAC bugs do not always cause problems in real execution and are hard to reproduce at runtime. In this example, the function *dma_pool_alloc* called with *GFP_KERNEL* only sleeps when memory is insufficient. Even in a low-memory situation, this SAC bug is not always triggered at runtime in a multi-core system, because of the non-determinism of concurrent execution. (3) Multiple layers of function calls need to be considered when finding SAC bugs. In this example, the function *dma_pool_alloc* is called across two function levels after *spin_lock_irqsave* is called.

The bug in Figure 1 has been fixed, but many SAC bugs still remain in current kernel modules. Some recent studies [12, 48] have shown that SAC bugs have caused serious system hangs, and these bugs were often hard to locate and reproduce. Thus, to improve the reliability of the operating system, it is necessary to design an approach to detect SAC bugs in kernel modules.

3. Challenges and Techniques

In this section, we first discuss the main challenges in detecting SAC bugs and then propose our techniques to address these challenges.

3.1 Challenges and Overview of Our Solutions

There are four main challenges in detecting SAC bugs in kernel modules:

C1: *Code analysis coverage, accuracy and time.* A key goal in bug detection is to efficiently cover more code and generate accurate results. Running kernel modules can be difficult (for example, running a driver needs the associated device), and thus we use static analysis to achieve high code coverage without the need to execute the code. Static analysis can be either flow-sensitive or flow-insensitive. Flow-sensitive analysis searches each code path of a branch and can cover all code paths. For this reason, it can produce accurate results, but it often requires much time and memory especially in inter-procedural analysis. Flow-insensitive analysis handles each code line instead of each path. Thus, it is more efficient, but its results may be less accurate. We propose a *hybrid flow analysis* to obtain the advantages of both flow-sensitive and -insensitive analysis. It uses flow-sensitive analysis when its accuracy is expected to be beneficial and falls back to flow-insensitive analysis when full accuracy is not necessary. We will introduce the hybrid flow analysis in Section 3.2.1.

C2: *Sleep-able function extraction.* Determining whether a function can sleep often requires a good understanding of the kernel code. Specifically, for a function defined in the kernel module (referred to as a *module function* subsequently), whether it can sleep depends on whether the called kernel interfaces can sleep. Using this idea, we design a *heuristics-based* method that first collects all kernel interfaces possibly called in atomic context of the kernel module, and then analyzes the kernel source code and comments to identify sleep-able ones. We will introduce this method in Section 3.2.2.

C3: *Filtering out repeated and false bugs.* Some detected bugs may be repeated, because they take the spinlock at the same place and call the same sleep-able function, but only differ in their code paths. Moreover, some detected bugs may be false positives, as the analysis does not consider variable value information, and thus may search some infeasible code paths. We design a *path-check* method that checks the code path of each detected bug to filter out repeated reports and false bugs. We will introduce it in Section 3.2.3.

C4: *Bug fixing recommendation.* After finding real bugs, the user may manually write patches to fix them. Besides, incorrect patches can introduce new bugs [21]. To reduce the manual work of bug fixing, we summarize common patterns for fixing SAC bugs, and propose a *pattern-based* method to automatically generate recommended patches to help fix the bugs. We will introduce this method in Section 3.2.4.

3.2 Key Techniques

3.2.1 Hybrid Flow Analysis

Our hybrid flow analysis is used to identify the code in atomic context. It is based on two points: (1) The analysis is context-sensitive and inter-procedural, in order to maintain the spinlock status and detect atomic context across functions calls. (2) The choice of flow-sensitive or -insensitive analysis is made as follows: *if a module function calls a spin-lock or spin-unlock function (this module function is referred to as a target function) or it is called by an interrupt handler, flow-sensitive analysis is used to analyze each code path from the entry basic block; otherwise, flow-insensitive analysis is used to handle each function call made by the function.* In the first case, flow-sensitive analysis is used to accurately maintain the spinlock status and collect code paths for subsequent bug filtering. In the second case, flow-insensitive analysis is used to reduce analysis cost, because in this case, the spinlock status is expected not to change explicitly.

Our hybrid flow analysis has two steps. The first step identifies target functions and interrupt handler functions, as flow-sensitive analysis is performed in these functions. For target functions, we analyze the definition of each module function and check whether it calls a spin-lock or spin-unlock function. For interrupt handler functions, we identify the calls to interrupt-handler-register kernel interfaces (like *request_irq* in the Linux kernel), and extract interrupt handler functions from the related arguments.

The second step performs the main analysis. Figure 2 presents the procedure *FlowAnalysis*. It maintains two stacks, namely a path stack (*path_stack*) to store the executed code path and a lock stack (*lock_stack)* to store the spinlock status. A flag (*g_intr_flag*) is used to indicate whether the code is in an interrupt handler. If *lock_stack* is not empty or *g_intr_flag* is *TRUE*, the code is in atomic context. *FlowAnalysis* uses *HanCall* to handle a function call and *HanBlock* to handle a basic block. We introduce them as follows:

HanCall. It handles the function call *mycall* with the arguments *path_stack* and *lock_stack*, to check if the definition of the function called by *mycall* needs to be handled, and if so to determine if the flow-sensitive or -insensitive analysis should be used. Firstly, *HanCall* checks if *lock_stack* is empty and *g_intr_flag* is *FALSE* (lines 1-3). If so, no spinlock is held and the code is not in an interrupt handler, and thus *HanCall* returns. Secondly, *HanCall* uses *path_stack* to check if *mycall* has been analyzed (lines 4-6). If so, it returns to avoid repeated analysis. Note that this prevents infinite looping on recursive calls. Thirdly, *HanCall* adds *mycall* into *path_stack*, and gets the called function *myfunc* (lines 7-8). Fourthly, *HowToFunc* (line 9) performs the analyses presented in Sections 3.2.2 and 4.1.3. Fifthly, *HanCall* checks if *myfunc* is a module function (lines 10-12). If not, it returns. Finally, it handles the definition of *myfunc* (lines 13-20). If *myfunc* is a target function or *g_intr_flag* is *TRUE*, flow-sensitive analysis is used to handle its entry basic block using *HanBlock* (lines 13-

```
HanCall(mycall, path_stack, lock_stack)
 1: if lock_stack == ø and g_intr_flag == FALSE then
 2:     return;
 3: end if
 4: if PathHasExisted(mycall, path_stack) == TRUE then
 5:     return;
 6: end if
 7: AddPathStack(mycall, path_stack);
 8: myfunc := GetCalledFunction(mycall);
 9: HowToFunc(myfunc, path_stack, lock_stack, g_intr_flag);
10: if IsModuleFunc(myfunc) == FALSE then
11:     return;
12: end if
13: if IsTargetFunc(myfunc) == TRUE or g_intr_flag == TRUE then
14:     entry_block := GetEntryBlock(myfunc);
15:     HanBlock(entry_block, path_stack, lock_stack);
16: else
17:     foreach call in FunctionCallList(myfunc) do
18:         HanCall(call, path_stack, lock_stack);
19:     end foreach
20: end if
```

```
HanBlock(myblock, path_stack, lock_stack)
 1: if PathHasExisted(myblock, path_stack) == TRUE then
 2:     return;
 3: end if
 4: AddPathStack(myblock, path_stack);
 5: foreach func_call in FunctionCallList(myblock) do
 6:     if func_call is a call to a spin-lock function then
 7          Push func_call onto lock_stack;
 8:     else if func_call is a call to a spin-unlock function then
 9:         Pop an item from lock_stack;
10:     else
11:         HanCall(func_call, path_stack, lock_stack);
12:     end if
13: end foreach
14: if lock_stack == ø and g_intr_flag == FALSE then
15:     return;
16: end if
17: foreach block in SuccessorBlocks(myblock) do
18:     HanBlock(block, path_stack, lock_stack);
19: end foreach
```

```
FlowAnalysis: Main hybrid flow analysis
 1: foreach func in target_func_set do
 2:     lock_block_set := GetLockBlockSet(func);
 3      foreach block in lock_block_set do
 4:         path_stack := ø; lock_stack := ø; g_intr_flag := FALSE;
 5:         HanBlock(block, path_stack, lock_stack);
 6:     end foreach
 7: end foreach
 8: foreach func in intr_handler_func_set do
 9:     path_stack := ø; lock_stack := ø; g_intr_flag := TRUE;
10:     entry_block := GetEntryBlock(func);
11:     HanBlock(entry_block, path_stack, lock_stack);
12: end foreach
```

Figure 2: Hybrid flow analysis.

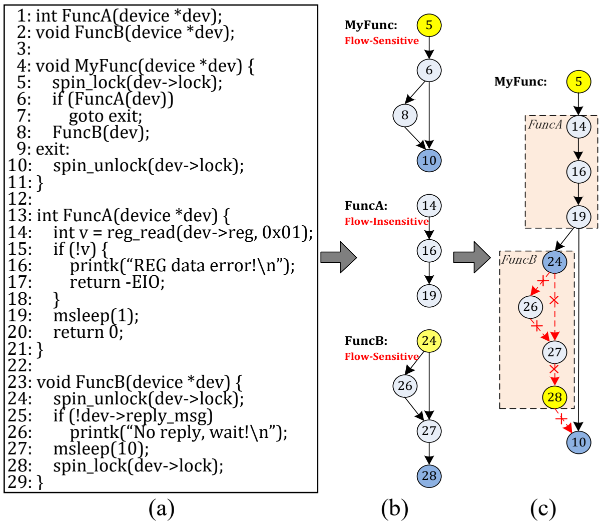
15); otherwise, flow-insensitive analysis is used to handle each function call made by *myfunc* using *HanCall* (lines 16-20).

HanBlock. It handles the basic block *myblock* with the arguments *path_stack* and *lock_stack*, to perform flow-sensitive analysis as well as maintain the spinlock status. Firstly, *HanBlock* uses *path_stack* to check if *myblock* has been analyzed (lines 1-3). If so, it returns to avoid repeated analysis. Secondly, *HanBlock* adds *myblock* into *path_stack* (line 4). Thirdly, *HanBlock* handles each function call in *myblock* (lines 5-13). If the function call is a call to a spin-lock or spin-unlock function, *HanBlock* pushes the call onto or pops an item from *lock_stack*; otherwise, the call is handled by *HanCall*. Fourthly, *HanBlock* checks if *lock_stack* is empty and *g_intr_flag* is *FALSE*. If so, it returns (lines 14-16); otherwise, each successive basic block of *myblock* is handled using *HanBlock* (lines 17-19).

FlowAnalysis. It performs the main analysis, in two steps. Firstly, each target function is analyzed (line 1-7). For a target function, each basic block that contains a spin-lock function call is an analysis entry. In this case, *path_stack* and *lock_stack* are first set to empty, and *g_intr_flag* is set to *FALSE*. Then, the analysis is started by using *HanBlock* to handle this basic block. Secondly, each interrupt handler function is analyzed (line 8-12). In this case, *path_stack* and *lock_stack* are set to empty, but *g_intr_flag* is set to *TRUE*. Then, the analysis is started by using *HanBlock* to handle the entry basic block of the interrupt handler function.

Our hybrid flow analysis has three main advantages: (1) The functions that are possibly called in atomic context can be accurately detected; (2) Detailed code paths and complete spinlock status are maintained, to help accurately detect atomic context; (3) Many unnecessary paths are not considered to reduce the analysis time. However, a main limitation of our analysis is that variable value information is not considered, which may cause false positives in bug detection.

We illustrate our hybrid flow analysis using some simplified driver-like code shown in Figure 3. As shown in Figure 3(a), the module consists of *MyFunc*, *FuncA* and *FuncB*, where *MyFunc* calls *FuncA* and *FuncB*. Because *MyFunc* and *FuncB* both call *spin_lock* and *spin_unlock*, they are target functions and handled by the flow-sensitive analysis; because *FuncA* does not call spin-lock or spin-unlock functions, it is handled by the flow-insensitive analysis. Figure 3(b) presents the call path of each function, with the code line numbers from Figure 3(a) shown in the vertices. Figure 3(c) shows the call paths used in inter-procedural analysis of *MyFunc*. During the analysis, no spinlock is held after the first line of *FuncB* (line 24), thus the following call paths in *FuncB* are not analyzed. In total, only two useful call paths marked in solid edges in Figure 3(c) are handled when analyzing *MyFunc*, and they are the only necessary call paths for atomic context analysis in this case.



```
 1: int FuncA(device *dev);
 2: void FuncB(device *dev);
 3:
 4: void MyFunc(device *dev) {
 5:     spin_lock(dev->lock);
 6:     if (FuncA(dev))
 7:         goto exit;
 8:     FuncB(dev);
 9: exit:
10:     spin_unlock(dev->lock);
11: }
12:
13: int FuncA(device *dev) {
14:     int v = reg_read(dev->reg, 0x01);
15:     if (!v) {
16:         printk("REG data error!\n");
17:         return -EIO;
18:     }
19:     msleep(1);
20:     return 0;
21: }
22:
23: void FuncB(device *dev) {
24:     spin_unlock(dev->lock);
25:     if (!dev->reply_msg)
26:         printk("No reply, wait!\n");
27:     msleep(10);
28:     spin_lock(dev->lock);
29: }
```

Figure 3: Example of hybrid flow analysis.

3.2.2 Heuristics-Based Sleep-able Function Extraction

We use some heuristics to accurately extract sleep-able kernel interfaces in the kernel modules. Firstly, we perform our hybrid flow analysis on the analyzed kernel module(s), to collect all kernel interfaces that are possibly called in atomic context, through *HowToFunc* in Figure 2. The collected information is stored into a database as *intermediate results*, including the function name, constant arguments, file name and so on. Secondly, we use some heuristics to inter-procedurally analyze the call graph of each collected kernel interface, and determine whether it can sleep. If a kernel interface satisfies one of the five criteria, we identify it sleep-able:

- It calls a basic sleep-able function, like *schedule* in the Linux kernel and *sleep* in the NetBSD kernel.
- It is called with a specific constant argument indicating it can sleep, like *GFP_KERNEL* in the Linux kernel and *M_WAITOK* in the FreeBSD kernel.
- It calls a specific macro that indicates the operation can sleep, like *might_sleep* in the Linux kernel.
- The comments in or before it contain keywords like *"can sleep"* and *"may block"*.
- It calls an already identified sleep-able kernel interface in the call graph.

To avoid repeated checking, we maintain two cache lists. If a function is marked as sleep-able, it is added to a *sleep-able* list; otherwise it is added to a *non-sleep* list. When analyzing a function, we first check whether the function is in either of these lists.

After the extraction, we get the sleep-able kernel interfaces that are possibly called in atomic context of the analyzed kernel modules(s). These kernel interfaces can be used to detect SAC bugs in the kernel module(s).

```
FILE: linux-4.11.1/drivers/scsi/ufs/ufshcd.c
504. static int ufshcd_wait_for_register(..., bool can_sleep) {
     ......
515.   if (can_sleep)
516.     usleep_range(...);
517.   else
518.     udelay(...);
     ......
527. }
```

(a) Checking a variable

```
FILE: linux-4.11.1/drivers/block/DAC960.c
783. static void DAC960_ExecuteCommand(...) {
     ......
794.   if (in_interrupt())
795.     return;
796.   wait_for_completion(...);
797. }
```

(b) Checking the return value of a kernel interface

Figure 4: Examples of path checks in drivers.

3.2.3 Path-Check Bug Filtering

We use the detailed code paths recorded in our hybrid flow analysis to filter out repeated and false SAC bugs.

Firstly, we filter out repeated bugs. For each new possible bug, we check whether its entry and terminal basic blocks are the same as those of an already detected bug, and whether they call the same sleep-able kernel interface. If both conditions are satisfied, this possible bug is marked as a repeated bug and is filtered out.

Secondly, we filter out false bugs, which are mainly introduced by the fact that our hybrid flow analysis neglects variable value information. The best strategy is to validate path conditions [6]. But it is often hard to ensure the accuracy and efficiency when control flow is complex, especially across function calls.

By studying the Linux kernel source code, we find a useful and common semantic information for variables: *a conditional that checks a parameter of the containing function or the return value of a specific kernel interface is often used to decide whether sleeping is allowed.* Figure 4 presents two examples in Linux driver code. In Figure 4(a), a conditional checks the function parameter *can_sleep* to decide whether the sleep-able kernel interface *usleep_range* can be called. In Figure 4(b), a conditional checks the return value of the kernel interface *in_interrupt* to check whether the code is executed in an interrupt handler to decide whether the sleep-able kernel interface *wait_for_completion* can be called. Using this semantic information, we design a straightforward strategy to cover common cases. If the code path of a possible bug satisfies one of the two criteria, we mark this bug as a false bug and filter it out:

- The path contains a conditional that checks a parameter of the containing function, and the name of this parameter contains a keyword like *"can_sleep"*, *"atomic"* and *"can_block"*.
- The path contains a conditional that checks the return value of a kernel interface used to check atomic context, like *in_interrupt* in the Linux kernel.

We propose a path-check method that uses the above steps, to automatically and effectively filter out repeated reports and false bugs.

3.2.4 Pattern-based Patch Generation

By studying Linux kernel patches, we have found four common patterns of fixing SAC bugs:

P1: Replace the sleep-able kernel interface with a non-sleep kernel interface having the same functionality, like *usleep_range* ⇒ *udelay* in Figure 4(a).

P2: Replace the specific sleep-able constant flag with a non-sleep flag, like *GFP_KERNEL* ⇒ *GFP_ATOMIC* in Figure 1.

P3: Move the sleep-able kernel interface to some place where a spinlock is not held.

P4: Replace the spinlock with a lock that allows sleeping, like *spin_lock* ⇒ *mutex_lock* and *spin_unlock* ⇒ *mutex_unlock* in the Linux kernel.

These patterns have different usage scenarios and raise different challenges. Firstly, P1 and P2 can be used for all atomic contexts, while P3 and P4 are only used when holding a spinlock. Secondly, P1 and P2 involve simple modifications, while P3 and P4 involve more difficult modifications and are error-prone. Using P3 requires carefully determining where the sleep-able function should be moved to. Using P4 requires modifying all locking and unlocking operations. Thus, using P3 and P4 to automatically generate patches is hard.

We only use P1 and P2 to automatically generate recommended patches, because these patterns are simple and effective. Supporting P3 and P4 is left as future work. The method has three steps. Firstly, the bug is located using its code path, and the relevant fixing pattern (P1 or P2) is selected according to the code. If no relevant pattern is available, no patch is generated. Secondly, the buggy code is corrected by using the selected pattern. Finally, a patch is generated by comparing the corrected code to original code.

This pattern-based method has two advantages. Firstly, it can reduce the manual work of bug fixing. Secondly, by using common fixing patterns, it can ensure the correctness of the generated patches.

4. Approach

Based on the four techniques in Section 3.2, we propose a static approach DSAC, to effectively detect SAC bugs in kernel modules and recommend patches to help fix the detected bugs. We have implemented DSAC with the Clang compiler [49], and perform static analysis on the LLVM bytecode of the kernel module. Figure 5 presents the architecture of DSAC, which has five parts:

- ***Code compiler.*** For a given kernel module, this part compiles all the source files of the kernel module into a single LLVM bytecode file.

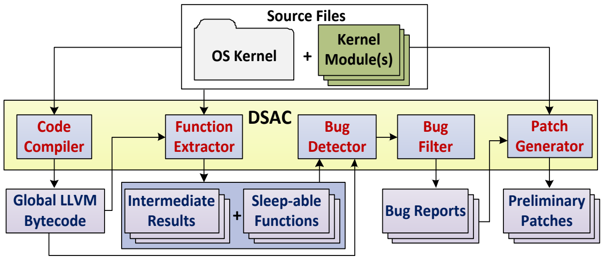


Figure 5: Overall architecture of DSAC.

- ***Function extractor.*** With the LLVM bytecode and the kernel source code, this part uses our hybrid flow analysis and heuristics-based method to generate intermediate results and extract sleep-able kernel interfaces called in the kernel module(s).
- ***Bug detector.*** With the extracted sleep-able kernel interfaces and intermediate results, this part reuses our hybrid flow analysis to automatically detect possible SAC bugs from the LLVM bytecode.
- ***Bug filter.*** This part uses our path-check method to filter out repeated and false bugs and generates reports for the final detected SAC bugs.
- ***Patch generator.*** With the bug code paths and kernel module source code, this part uses our pattern-based method to automatically recommend patches to help fix the bugs.

Based on the architecture, DSAC consists of four phases which are introduced as follows.

4.1 Function Information Collection

In this phase, DSAC performs two steps:

Firstly, the code compiler compiles each source file of the kernel module into a LLVM bytecode file, and then links all bytecode files into a single bytecode file. This single bytecode file includes all module function definitions, thus all analyses of the kernel module can be directly performed on only this single bytecode file.

Secondly, the function extractor performs the hybrid flow analysis to collect the information about functions that are possibly called when holding a spinlock or in an interrupt handler. The information is stored in a MySQL [52] database as the intermediate results, including the function name, constant arguments, file name, etc. The intermediate results will be later used in sleep-able kernel interface extraction and bug detection.

4.2 Sleep-able Kernel Interface Extraction

In this phase, the function extractor first extracts function call graphs of kernel interfaces and comments of these kernel interfaces, and then uses the heuristics-based method to extract sleep-able kernel interfaces. The user can check and modify the extracted sleep-able kernel interfaces as needed.

4.3 Bug Detection

In this phase, DSAC first detects possible SAC bugs, and then filters out repeated reports and false bugs.

Firstly, the bug detector uses the hybrid flow analysis to check whether each extracted sleep-able kernel interface is called in atomic context, which is implemented in *HowToFunc* in Figure 2. If so, a possible bug and its detailed code path to the sleep-able kernel interface call are recorded. To speed up analysis, we use the intermediate results to only check the buggy kernel modules.

Secondly, the bug filter filters out repeated reports and false bugs. Finally, DSAC produces detailed reports for the found bugs (including code paths and source file names), so the user can locate and check the bugs.

4.4 Recommended Patch Generation

In this phase, the patch generator automatically generates recommended patches to help fix the bugs. Then, the user can use the detailed code paths found in the bug reports to write log messages, and finally submit these patches to kernel maintainers.

5. Evaluation

5.1 Experimental Setup

To validate the effectiveness of DSAC, we first evaluate it on Linux drivers, which are typical kernel modules. To cover different kernel versions, we select an old version 3.17.2 (released in October 2014), and a new version 4.11.1 (released in May 2017). Then, to validate the generality of DSAC, we use it to check file systems and network modules in the Linux kernel. Finally, to validate the portability of DSAC, we run it in FreeBSD and NetBSD to check their kernel code.

We run the experiments on a Lenovo x86-64 PC with four Intel i5-3470@3.20G processors and 4GB memory. We compile the code using Clang 3.2. We use the kernel configuration *allyesconfig* to enable all drivers, file systems and network modules for the *x86* architecture.

To run DSAC, the user performs three steps. Firstly, the user configures DSAC for the checked OS kernel, by providing the names of spin-lock and -unlock functions (such as *spin_lock_irq* and *spin_unlock_irq* for the Linux kernel), interrupt-handler-register functions (such as *request_irq* for the Linux kernel), and basic sleep-able kernel interfaces (such as *schedule* for the Linux kernel). Secondly, the user compiles the source code of the kernel modules and OS kernel using the kernel's underlying *Makefile* and DSAC's compiling script. As a result, DSAC produces sleep-able functions and intermediate results. Finally, the user executes DSAC's bug-detecting script to detect bugs and generate recommended patches. The second and third steps are fully automated.

Table 1: Results of extracting sleep-able kernel interfaces.

Description		3.17.2	4.11.1
Code handling	Handled bytecode files	3377	4396
	Source files (.c)	8321	11153
	Source code lines	7392K	9464K
Hybrid flow analysis	Entry basic blocks	32167	37770
	Handled INTR functions	578	673
Heuristic extraction	Recorded functions	3104	3613
	Sleep-able kernel interfaces	70 (51)	94 (63)
Time usage	Original compilation	47m21s	55m34s
	DSAC total	108m43s	129m58s
	DSAC pure	61m22s	74m22s

Table 2: Detected bugs and generated patches in drivers.

Description		3.17.2	4.11.1
Detected bugs	Repeated filtered	479630	629924
	False filtered	282	430
	Final detected	215	340
	Interrupt handling	7	17
	Real	200	320
Patch generation	P1 (replace the function)	-	28 (18)
	P2 (replace the flag)	-	15 (12)
	Total	-	43 (30)
Time usage	Bug detection	6m31s	8m46s
	Patch generation	-	1m02s
	Total	6m31s	9m48s

5.2 Extracting Sleep-able Kernel Interfaces

We first extract the sleep-able kernel interfaces that are called in atomic context of the drivers. Table 1 presents the results for Linux 3.17.2 and 4.11.1. We make the following observations:

1) DSAC can scale to large code bases. It handles 7M and 9M source code lines from 8K and 11K source files. And the analysis is started from many entry basic blocks and many interrupt handler (INTR) functions.

2) Our heuristics-based method can efficiently extract real sleep-able kernel interfaces that are called in atomic context of the analyzed drivers. In Linux 3.17.2 and 4.11.1, 70 and 94 sleep-able kernel interfaces are respectively identified from among 3104 and 3613 different kernel interfaces (candidate functions) that are possibly called in atomic context. We manually check the kernel interfaces identified as sleep-able, and find that all of them can sleep at runtime. Over 97% of the candidate functions are automatically filtered out, thus the manual work of checking these functions is saved.

3) Our code analysis is efficient. DSAC respectively spends around 108 and 129 minutes on handling 8K and 11K driver source files, including the compilation time of these source files using the Clang compiler. Excluding compilation time, DSAC spends 61 and 74 minutes respectively, amounting to less than 0.44 seconds per source file.

4) Many of the extracted sleep-able kernel interfaces are related to resource handling (such as allocation and release). The data in parentheses present the number of these kernel interfaces, which amount to more than 60% of all the sleep-able kernel interfaces.

5.3 Detecting Bugs and Generating Patches

Based on the above extracted sleep-able kernel interfaces, we use DSAC to perform bug detection and recommend patches. Firstly, to validate whether DSAC can find known bugs, we use DSAC to check the drivers in Linux 3.17.2. We do not generate patches in this case, because this kernel version is very old. Secondly, to validate whether DSAC can find new bugs and recommend patches to help fix them, we use DSAC to check the drivers in Linux 4.11.1. We count the bugs according to the pair of entry and terminal basic blocks. To check results' accuracy, we manually check all detected bugs to identify whether they are real bugs. Table 2 shows the results. We have the following findings:

1) Our path-check filtering method is effective in automatically filtering out repeated reports and false bugs.

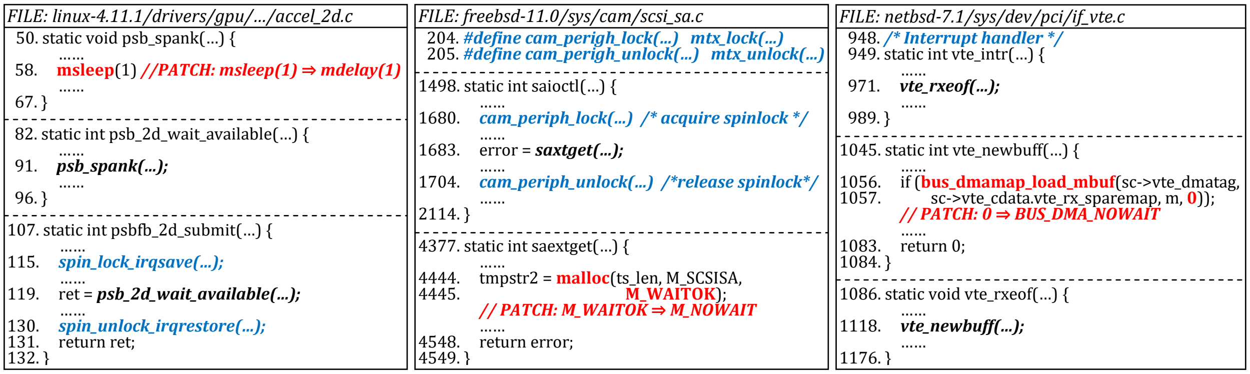
2) Of the 215 bugs reported by DSAC in the drivers of Linux 3.17.2, we have identified 200 as real bugs, 50 of which have been fixed in Linux 4.11.1. By reading the messages in relevant Linux driver mailing lists, we find that kernel maintainers confirmed that these fixed bugs could cause serious problems, like system hangs. The results indicate DSAC can indeed find known bugs.

3) Of the 340 bugs reported by DSAC in the drivers of Linux 4.11.1, we have identified 320 as real bugs. 150 bugs are inherited from the legacy code in 3.17.2, and 170 bugs are introduced by new functionalities and new drivers. We have reported all the bugs that we identified as real to kernel maintainers. As of January 2018, 209 bugs have been confirmed, and replies for the other bugs have not been received. The results indicate DSAC can indeed find new real bugs.

4) DSAC can accurately find real bugs in our evaluated driver code. The false positive rates are respectively only 7.0% and 5.9% in the drivers of Linux 3.17.2 and 4.11.1, based on our identification of real bugs. Reviewing the driver source code, we find these false positives are mainly introduced by the fact that some invalid code paths are searched by our hybrid flow analysis and our path-check method does not filter them out.

5) Few of the detected bugs are in interrupt handlers (7 bugs in 3.17.2, and 17 bugs in 4.11.1). Indeed, driver developers often write clear comments to mark the driver functions that are called from an interrupt handler, to prevent calling sleep-able functions in these functions.

6) DSAC automatically and successfully generates 43 patches that it recommends to help fix 82 real bugs in Linux 4.11.1. Table 2 classifies the patches by the pattern in Section 3.2.4 that is used. We manually review these patches, add appropriate descriptions, and then submit them to the relevant kernel maintainers. As of January 2018, 30 patches have been applied, noted in



```
FILE: linux-4.11.1/drivers/gpu/.../accel_2d.c
 50. static void psb_spank(...) {
         ......
 58.     msleep(1) //PATCH: msleep(1) ⇒ mdelay(1)
         ......
 67. }
---------------------------------------------
 82. static int psb_2d_wait_available(...) {
         ......
 91.     psb_spank(...);
         ......
 96. }
---------------------------------------------
107. static int psbfb_2d_submit(...) {
         ......
115.     spin_lock_irqsave(...);
         ......
119.     ret = psb_2d_wait_available(...);
         ......
130.     spin_unlock_irqrestore(...);
131.     return ret;
132. }
```

(a) Linux *gma500* driver

```
FILE: freebsd-11.0/sys/cam/scsi_sa.c
 204. #define cam_perigh_lock(...)   mtx_lock(...)
 205. #define cam_perigh_unlock(...)   mtx_unlock(...)
---------------------------------------------
1498. static int saioctl(...) {
          ......
1680.     cam_periph_lock(...)  /* acquire spinlock */
          ......
1683.     error = saxtget(...);
          ......
1704.     cam_periph_unlock(...)  /*release spinlock*/
          ......
2114. }
---------------------------------------------
4377. static int saextget(...) {
          ......
4444.     tmpstr2 = malloc(ts_len, M_SCSISA,
4445.                            M_WAITOK);
          // PATCH: M_WAITOK ⇒ M_NOWAIT
          ......
4548.     return error;
4549. }
```

(b) FreeBSD *scsi_sa* driver

```
FILE: netbsd-7.1/sys/dev/pci/if_vte.c
 948. /* Interrupt handler */
 949. static int vte_intr(...) {
          ......
 971.     vte_rxeof(...);
          ......
 989. }
---------------------------------------------
1045. static int vte_newbuff(...) {
          ......
1056.     if (bus_dmamap_load_mbuf(sc->vte_dmatag,
1057.             sc->vte_cdata.vte_rx_sparemap, m, 0));
          // PATCH: 0 ⇒ BUS_DMA_NOWAIT
          ......
1083.     return 0;
1084. }
---------------------------------------------
1086. static void vte_rxeof(...) {
          ......
1118.     vte_newbuff(...);
          ......
1176. }
```

(c) NetBSD *if_vte* driver

Figure 6: Examples of the real bugs detected by DSAC.

```
********** BUG  **********
Sleep-able function: msleep
[FUNC] psb_spank (drivers/gpu/.../accel_2d.c: LINE 58)
[FUNC] psb_2d_wait_available (drivers/gpu/.../accel_2d.c: LINE 91)
......
[FUNC] psbfb_2d_submit (drivers/gpu/.../accel_2d.c: LINE 119)
......
[FUNC] psbfb_2d_submit (drivers/gpu/.../accel_2d.c: LINE 115)
```

Table 3: Bug distribution according to driver class.

Driver Class	scsi	network	staging	gpio	others
Bugs	103 (32%)	84 (26%)	62 (19%)	12 (4%)	59 (18%)

parentheses in Table 2. 2 patches were not directly applied as the maintainers wanted to fix the bugs in other ways (such as P3 and P4). There has been no reply yet for 11 patches. There are still 238 real bugs for which DSAC cannot recommend patches, as they do not match P1 or P2. Most of these bugs can be fixed using P3 or P4. But those patterns require more difficult changes, and DSAC is not currently able to automatically apply them. In general, the results indicate that DSAC can generate a number of correct patches to reduce the manual work of bug fixing.

7) Bug detection and patch generation are efficient, requiring less than 10 minutes. The reasons include that intermediate results are used to reduce repeated analysis and our hybrid flow analysis is efficient.

Reviewing the results, we find two interesting things. Firstly, most of the detected bugs involve multiple functions. Indeed, driver developers may easily forget that the code is in atomic context across multiple function calls. Secondly, many of the detected bugs are related to resource allocation and release, because many extracted sleep-able functions relate to this issue.

We also classify the 320 real bugs found by DSAC in Linux 4.11.1 drivers, according to driver class. Table 3 shows the top results. We find that SCSI and network drivers share 58% of all bugs.

Table 4: Results of Linux *fs* and *net*, FreeBSD and NetBSD.

Description		**fs & net**	**FreeBSD**	**NetBSD**
Code handling	Handled bytecode files	925	632	710
	Source files	2506	1615	1977
	Source code lines	2013K	1759K	1896K
Function extraction	Recorded functions	1927	582	304
	Sleep-able kernel interfaces	34	12	10
Bugs & patches	Filtered bugs	682081	508	2414
	Final detected bugs	42	39	7
	Real bugs	39	35 (26)	7 (7)
	Generated patches	5	10	3
Pure time usages		32m45s	49m12s	43m38s

Figure 6(a) shows a real bug detected by DSAC in the *gma500* driver of Linux 4.11.1, which has been confirmed by the developer. The function *psbfb_2d_submit* first calls *spin_lock_irqsave* to acquire a spinlock (line 115), and then it calls *psb_2d_wait_available* (line 119) that calls *psb_spank* in definition (line 91). The function *psb_spank* calls *msleep* (line 58) that can sleep. To help fix the bug, our pattern-based method recommends a patch that replaces *msleep* with *mdelay* (P1), and this patch has been applied by the kernel maintainer. Part of the DSAC's report for this bug is listed above Table 3.

5.4 Generality and Portability

We use DSAC to check file systems and network modules in Linux 4.11.1. Then we run DSAC in FreeBSD 11.0 and NetBSD 7.1 to check their kernel code. Table 4 shows the results. We have the following findings:

1) DSAC works normally when checking Linux file systems and network modules and other OS kernels. DSAC can handle their source code in a modest amount of time. It can extract real sleep-able kernel interfaces and filter out many repeated reports and false bugs.

2) DSAC in total finds 81 real bugs out of the 88 detected bugs. The false positive rate is thus 8.0%. The false positives are again due to searching invalid code paths. As of January 2018, 63 of these bugs have been confirmed by kernel developers. Figure 6(b) and (c) present two real SAC bugs found by DSAC in FreeBSD *scsi_sa* and NetBSD *if_vte* drivers. These bugs involve respectively a spinlock and an interrupt handler.

3) DSAC in total generates 18 recommended patches to help fix 59 real bugs. We manually add appropriate descriptions and submit them to kernel maintainers. As of January 2018, 13 of the patches have been applied.

Reviewing the results, we find two interesting things. Firstly, compared to the Linux kernel, fewer SAC bugs are detected in FreeBSD and NetBSD. The main reason is that in FreeBSD and NetBSD, many kernel interfaces that can sleep are carefully designed to avoid SAC bugs. For example, the FreeBSD *msleep* function takes the held spinlock as an argument and unlocks the spinlock before actually sleeping and then locks it again. Secondly, in FreeBSD and NetBSD, most of the detected bugs are in drivers, as shown in the parentheses on "Real bugs" line of Table 4. It shows that drivers remain a significant cause of system failures [39].

5.5 Sensitivity Analysis

DSAC performs flow-insensitive analysis to reduce time usage in specific cases when doing so is expected to not affect accuracy, and also maintains a lock stack to accurately identify the code in atomic context. To show the value of these two techniques, we modify DSAC to remove each of them, and evaluate each modified tool on a typical SCSI driver *fnic (drivers/scsi/fnic/)* of Linux 4.11.1. Original DSAC checks the driver in three seconds, and finds two real confirmed SAC bugs.

Flow-insensitive analysis. We use a full flow-sensitive analysis rather than the hybrid flow analysis. It finds the two SAC bugs too, but it spends two minutes, which is much longer than original DSAC.

Lock stack. We only keep a single bit indicating whether a lock is held rather than the lock stack during analysis. It also spends three seconds, but does not find any bugs. Indeed, the two bugs exist when two spinlocks are held and just one spinlock has been released, thus keeping a single bit cannot identify this atomic context.

5.6 Summary of Results

Our experiments show three significant results of using DSAC on the Linux, FreeBSD and NetBSD kernels:

- 401 new real bugs are found, of which 272 have been confirmed by kernel developers.
- Only 27 reports are false positives. Thus the overall false positive rate of bug detection is only 6.3%.
- 61 recommended patches are generated, of which 43 have been applied by kernel maintainers.

6. Comparison to Previous Approaches

Several previous approaches [2, 9, 16, 34] have considered SAC bugs. Among them, we select the *BlockLock* checker [34] to make a detailed comparison. We select this approach because: (1) It is a state-of-the-art tool to detect SAC bugs in the Linux kernel. (2) It is open-source and its bug reports are available [54]. In design, DSAC has some key improvements over *BlockLock*:

Code analysis. *BlockLock* only uses one bit of context information to check if a lock is held, so it may not accurately identify the code in atomic context when multiple locks are taken but only some of them are released. DSAC maintains a complete lock stack and performs context-sensitive analysis, thus it can accurately detect all code in atomic context. *BlockLock* is also not sensitive to the module *Makefile*, and thus may choose the wrong definition when unfolding a function call if the called function has multiple definitions. DSAC uses the module *Makefile* to accurately identify the definition of each function. And DSAC can detect SAC bugs in interrupt handlers and involving sleeping operations other than a call to an allocation function with *GFP_KERNEL*, which are not considered by *BlockLock*.

Sleep-able function extraction. *BlockLock* regards all functions called in the kernel as candidate functions to extract sleep-able functions. This strategy entails checking each function in the kernel inter-procedurally, so it may require much time. DSAC only treats the kernel interfaces possibly called in atomic context of the analyzed kernel module(s) as candidate functions, and skips the other functions not called in atomic context.

False bug filtering. *BlockLock* does not consider variable value information to validate path conditions, which may cause a number of false positives. DSAC checks the detailed code path of each possible bug, and filters out false bugs using useful and common semantic information for variables in atomic context.

Patch generation. *BlockLock* only reports bugs, but it does not help fix the bugs. DSAC uses common fixing patterns to generate recommended patches to help fix the bugs. The produced code paths of the bugs are also useful to help the user write log messages in the patches.

We also compare the results of *BlockLock* and DSAC, with two steps. Firstly, we download the bug reports of *BlockLock* on Linux 2.6.33 drivers, and get 49 reported bugs. We select the bugs related to the *x86* architecture based on driver *Kconfig* files. We get 31 reported SAC bugs (25 real bugs and 6 false bugs). Secondly, we use DSAC to check the Linux 2.6.33 driver source code. We use the kernel configuration *allyesconfig* to enable all drivers for the *x86* architecture. DSAC reports 42 sleep-able kernel interfaces and 228 reported SAC bugs. We manually check the bugs and find that 208 are real.

By manually comparing the bug reports shows: (1) 53 real bugs reported by DSAC are equivalent to 23 real bugs reported by *BlockLock*. DSAC reports more bugs because it detects sleep-able kernel interfaces, while *BlockLock* detects sleep-able functions. Thus, if a function defined in the kernel module calls several sleep-able kernel interfaces in atomic context, DSAC reports all these kernel interfaces, while *BlockLock* only reports this function. The two remaining real bugs reported by *BlockLock* are missed by DSAC, as Clang-3.2 cannot successfully compile the related driver source code. (2)

DSAC filters out all false bugs reported by *BlockLock*. (3) DSAC reports 155 real bugs missed by *BlockLock*. Most of these bugs involve multiple source files, and *BlockLock* cannot handle them very precisely. And 18 bugs are related to interrupt handling, which is not considered by *BlockLock*. (4) The false positive rate of DSAC is 8.8%, which is lower than that of *BlockLock*.

However, compared to *BlockLock*, an important limitation of DSAC is that its results are specific to a single kernel configuration. *BlockLock* is based on Coccinelle [33], which does not compile the source code. Thus it can conveniently check all source files without any kernel configuration. DSAC is based on LLVM, which compiles the source code with a selected kernel configuration. Thus, the 18 bugs found by *BlockLock* for *non-x86* architectures are missed by DSAC.

7. Limitations and Future Work

DSAC still has some limitations. Firstly, DSAC analyzes LLVM bytecode in which macros are expanded, thus the user needs to configure DSAC in terms of expanded versions of the functions and constants that are defined by macros. We plan to introduce source code information to address this issue. Secondly, DSAC cannot handle function pointers. We plan to use alias analysis [22, 46] to analyze them. Thirdly, as is typical for static analysis, the path-check method cannot filter out all invalid code paths produced by the hybrid flow analysis, which can introduce false positives. We plan to improve our path-check method by checking path conditions more accurately. Finally, the bug-fixing patterns P3 and P4 need to be supported, e.g., we plan to add the analysis for other kinds of locks to support P4.

8. Related Work

8.1 Detecting Concurrency Bugs

Many approaches [7, 14, 19, 28, 32, 38, 42, 43] have been proposed to detect concurrency bugs in user-mode applications. Some of them [7, 19, 42] use dynamic analysis to collect and analyze runtime information to detect concurrency bugs. But the code coverage of dynamic analysis is limited by test cases. Others [14, 32, 38, 43] use static analysis to cover more code without running the tested programs. But static analysis often introduces false positives. Some approaches [8, 26, 28] combine static and dynamic analysis to achieve higher code coverage with fewer false positives. Even though DSAC uses static analysis, it also exploits complementary information such as semantic information for variables to check code paths to filter out false positives.

To improve OS reliability, some approaches [13, 15, 17, 18, 40, 41, 44] detect some kinds of concurrency bugs like data races, but they do not detect SAC bugs. Several approaches [2, 9, 16, 34, 53] can detect common kinds of OS kernel bugs, including SAC bugs. But they do not specifically target SAC bugs, thus they may miss many real bugs or report many of false positives. For example, *BlockLock* [34] has an overall false positive rate of 20%, while DSAC has a lower one of 6.3%, and it also misses some real bugs found by DSAC.

8.2 Checking API Rules

Checking API rules is a promising way of finding deep and semantic bugs in the OS kernel. Some approaches [3, 5, 30, 31] use specified and known API rules to statically or dynamically detect API misuses. For example, with known paired reference count management functions, RID [30] uses a summary-based inter-procedural analysis to detect reference counting bugs. To find implicit API rules, some approaches [4, 23, 24, 27, 37, 45, 47] do specification mining by analyzing source code [24, 27, 37, 47] or execution traces [4, 23, 45], and then use the mined API rules to detect violations.

Most of these approaches focus on the temporal rules of common API usages, such as resource acquiring and releasing pairs [37, 45] and error handling patterns [4, 24], but these approaches have not targeted SAC bugs.

8.3 Improving Kernel Module Architecture

To prevent concurrency bugs, several improved kernel module architectures have been proposed, typically for device drivers. The active driver architecture [1, 36] runs each driver in a separate thread, which can serialize access to the driver and eliminate the possibility of concurrency bugs. In this way, the driver works serially and does not need to use locks, thus many common concurrency bugs will never occur. The user-mode device-driver architecture [20, 25, 35] runs each driver in a separate user-mode process. This architecture protects the OS kernel against crashes caused by driver code.

These approaches have a main limitation, namely that the driver code must be manually rewritten.

9. Conclusion

In this paper, we have proposed DSAC, a static approach, to effectively detect SAC bugs and automatically recommend patches to help fix them. DSAC uses four key techniques: (1) a hybrid flow analysis to identify the code in atomic context; (2) a heuristics-based method to extract sleep-able kernel interfaces; (3) a path-check method to filter out repeated reports and false bugs; (4) a pattern-based method to automatically generate recommended patches to help fix the bugs. We have used DSAC to check the kernel code of Linux, FreeBSD and NetBSD, and find 401 new real bugs. As of January 2018, 272 of them have been confirmed, and 43 of the patches generated by DSAC have been applied.

References

[1] S. Amani, P. Chubb, A. F. Donaldson, A. Legg, K. C. Ong, L. Ryzhyk, and Y. Zhu. Automatic verification of active device drivers. In *ACM SIGOPS Operating System Review*, volume 48, pages 106-118, 2014.

[2] Z. Anderson, E. Brewer, J. Condit, R. Ennals, D. Gay, M. Harren, G. C. Necula, and F. Zhou. Beyond bug-finding: sound program analysis for Linux. In *Proceedings of the 11th International Workshop on Hot Topics in Operating Systems (HotOS)*, pages 1-6, 2007.

[3] J. J. Bai, H. Q. Liu, Y. P. Wang, and S. M. Hu. Runtime checking for paired functions in device drivers. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC)*, pages 407-414, 2014.

[4] J. J. Bai, Y. P. Wang, H. Q. Liu, and S. M. Hu. Mining and checking paired functions in device drivers using characteristics fault injection. In *Information and Software Technology*, volume 73, pages 122-133, 2016.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st European Conference on Computer Systems (EuroSys)*, pages 73-85, 2006.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 209-224, 2008.

[7] Y. Cai, J. Zhang, L. Cao, and J. Liu. A deployable sampling strategy for data race detection. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE)*, pages 810-821, 2016.

[8] L. Chew, and D. Lie. Kivati: fast detection and prevention of atomic violations. In *Proceedings of 5th European Conference on Computer Systems (EuroSys)*, pages 307-320, 2010.

[9] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th International Symposium on Operating Systems Principles (SOSP)*, pages 73-88, 2001.

[10] Jonathan Corbet. Atomic context and kernel API design. In *Linux Weekly News (LWN.net)*, 2008. *https://lwn.net/Articles/274695/.*

[11] J. Corbet, A. Rubini, and G. K. Hartman. Spinlocks and atomic context. In *Linux Device Drivers*, 3rd edition, pages 118-119, 2005.

[12] D. Cotroneo, R. Natella, and S. Russo. Assessment and improvement of hang detection in the Linux operating system. In *Proceedings of the 28th International Symposium on Reliable Distributed Systems (SRDS)*, pages 288-294, 2009.

[13] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, pages 166-177, 2015.

[14] J. Deshmukh, E. A. Emerson, and S. Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE)*, pages 480-491, 2009.

[15] D. Engler, and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)*, pages 237-252, 2003.

[16] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 1-16, 2000.

[17] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 151-162, 2010.

[18] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. SKI: exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 415-431, 2014.

[19] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, pages 215-228, 2011.

[20] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 168-178, 2008.

[21] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 55-64, 2010.

[22] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 29th International Conference on Programming Language Design and Implementation (PLDI)*, pages 249-259, 2008.

[23] C. LaRosa, L. Xiong, and K. Mandelberg. Frequent pattern mining for kernel trace data. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC)*, pages 880-885, 2008.

[24] J. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, and G. Muller. WYSIWIB: a declarative approach to finding API protocols and bugs in Linux code. In *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN)*, pages 43-53, 2009.

[25] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. T. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: achieved performance. In *Journal of Computer Science and Technology (JCST)*, volume 20, issue 5, pages 654-664, 2005.

[26] Q. Li, Y. Jiang, T. Gu, C. Xu, J. Ma, X. Ma, and J. Lu. Effectively manifesting concurrency bugs in Android apps. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 209-216, 2016.

[27] Z. Li, and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th International Symposium on Foundations of Software Engineering (FSE)*, pages 306-315, 2005.

[28] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian. DCatch: automatically detecting distributed concurrency bugs in cloud systems. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 677-691, 2017.

[29] R. Love. Interrupt context. In *Linux Kernel Development*, 3rd edition, page 122, 2010.

[30] J. Mao, Y. Chen, Q. Xiao, and Y. Shi. RID: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 531-544, 2016.

[31] C. Min, S. Kashyap, B. Lee, C. Song, T. Kim. Cross-checking semantic correctness: the case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 361-377, 2015.

[32] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the 27th International Conference on Programming Language Design and Implementation (PLDI)*, pages 308-319, 2006.

[33] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*, pages 247-260, 2008.

[34] N. Palix, G. Thomas, S. Saha, C. Calvès, G. Muller, and J. Lawall. Faults in Linux 2.6. In *ACM Transactions on Computer Systems (TOCS)*, volume 32, issue 2, pages 4:1-4:40, 2014.

[35] M. J. Renzelmann, and M. M. Swift. Decaf: moving device drivers to a modern language. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIX ATC)*, pages 1-14, 2009.

[36] L. Ryzhyk, Y. Zhu, and G. Heiser. The case for active device drivers. In *Proceedings of the 1st Aisa-Pacific Workshop on Systems (APSys)*, pages 25-30, 2010.

[37] S. Saha, J. P. Lozi, G. Thomas, J. Lawall, and G. Muller. Hector: detecting resource-release omission faults in error-handling code for systems software. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)*, pages 1-12, 2013.

[38] A. Santhiar, and A. Kanade. Static deadlock detection for asynchronous C# programs. In *Proceedings of the 38th International Conference on Programming Language Design and Implementation (PLDI)*, pages 292-305, 2017.

[39] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)*, pages 207-222, 2003.

[40] L. Tan, Y. Zhou, and Y. Padioleau. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 11-20, 2011.

[41] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 391-402, 2016.

[42] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, pages 253-264, 2010.

[43] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, pages 602-629, 2005.

[44] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent Linux device drivers. In *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE)*, pages 501-504. 2007.

[45] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of 28th International Conference on Software Engineering (ICSE)*, pages 282-291, 2006.

[46] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th International Symposium on Code Generation and Optimization (CGO)*, pages 218-229, 2010.

[47] I. Yun, C. Min, X. Si, Y. Jiang, T. Kim, and M. Naik. APISan: sanitizing API usages through semantic cross-checking. In *Proceedings of the 25th USENIX Security Symposium*, pages 363-378, 2016.

[48] Y. Zhu, Y. Li, J. Xue, T. Tan, J. Shi, Y. Shen, and C. Ma. What is system hang and how to handle it. In *Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 141-150, 2012.

[49] Clang compiler. *http://clang.llvm.org/.*

[50] Linux kernel document for memory allocation. *https://www.kernel.org/doc/htmldocs/kernel-api/API-kmalloc.html.*

[51] LLVM Compiler Infrastructure. *https://llvm.org/.*

[52] MYSQL database. *https://www.mysql.com/.*

[53] Syzkaller tool. *https://github.com/google/syzkaller/.*

[54] Website for “Faults in Linux: Ten years later”. *http://faultlinux.lip6.fr/.*

Coccinelle: 10 Years of Automated Evolution in the Linux Kernel

Julia Lawall
Sorbonne University/Inria/LIP6

Gilles Muller
Sorbonne University/Inria/LIP6

Abstract

The Coccinelle C-program matching and transformation tool was first released in 2008 to facilitate specification and automation in the evolution of Linux kernel code. The novel contribution of Coccinelle was that it allows software developers to write code manipulation rules in terms of the code structure itself, via a generalization of the patch syntax. Over the years, Coccinelle has been extensively used in Linux kernel development, resulting in over 6000 commits to the Linux kernel, and has found its place as part of the Linux kernel development process. This paper studies the impact of Coccinelle on Linux kernel development and the features of Coccinelle that have made it possible. It provides guidance on how other research-based tools can achieve practical impact in the open-source development community.

1 Introduction

Today, everyone uses the Linux kernel, whether on a mobile phone (86% of the smartphones sold in the first quarter of 2017 were running Android [22]), in the cloud (at the end of 2016, 92% of virtual machine instances on Amazon's Elastic Compute Cloud (EC2) were running Linux), or on a supercomputer (at the end of 2017, all of the top 500 supercomputers were running Linux [63]). To support these diverse computing environments, the size of the Linux kernel has been steadily growing, reaching 16.5 million lines of code in the recently released version 4.15 (Jan. 2018). Furthermore, the existing source code of the Linux kernel is continually changing, with around 13,000 commits per release recently, to improve security, performance or maintainability, as well as to provide support for new services such as new kinds of devices, file systems, or hardware architectures.

A stumbling block in this continual revision of the Linux kernel is that ultimately some developer has to modify the source code. Developers have limited time, may not fully understand what a given change entails, and are prone to making mistakes, particularly when changes affect many code sites, pervasively, across multiple kernel subsystems. These problems are further compounded by the fact that the Linux kernel has a widely dispersed and very diverse set of developers, ranging from core maintainers, with many years of experience, to occasional contributors, to developers of out-of-tree code who do not participate in the Linux kernel developer community. Indeed, while over 1700 developers have contributed to each recent release, in each case a third or more of these developers have contributed only a single patch to that release. A potential solution is to formally specify changes and automate them. To be used in practice, such an approach must fit with the background and habits of the developers themselves.

The Coccinelle C-program matching and transformation tool was first released in 2008 to facilitate specification and automation of the evolution of Linux kernel code [44]. Coccinelle was built around the observation that Linux kernel developers already have a precise notation for describing changes with which they are very familiar, the *patch* [36]. A patch is an extract of source code in which some lines are annotated with `-` or `+` indicating that the line should be removed or added, respectively. All contributions to the Linux kernel pass in the form of patches through mailing lists where they are commented on by other developers, and thus developers are used to seeing and understanding them. Exploiting this background of kernel developers, Coccinelle is designed around a domain-specific language (DSL), SmPL (Semantic Patch Language), for expressing changes in terms of an abstracted form of patch, referred to as a *semantic patch*. Unlike a patch, which is tied to specific lines and files in the source code, a single semantic patch can update all relevant locations in the entire code base.

Today, Coccinelle has been under development for 12 years. 59 semantic patches are part of the Linux kernel source tree, and over 6000 Linux kernel commits, including 900 from Linux kernel maintainers, use Coccinelle. Coccinelle retains as a guiding principle the notion of an abstracted patch. Nevertheless, it has grown, both in terms of expressiveness and to improve performance, based on lessons learned from the experiences of Linux developers and other users. At the same time, it has successfully integrated into the Linux kernel community.

Coccinelle has been used in previous research [30, 31, 34, 46]. This paper instead focuses on its evolution and impact. We examine its initial design (Sec. 2), and how that design has been refined in response to experience with the tool and feedback from users (Sec. 3). We then evaluate performance (Sec. 4) and the benefit of our expressivity extensions (Sec. 5), quantify the impact of Coccinelle in the Linux community (Sec. 6), and give an overview of its wider use (Sec. 7). Finally, we present some related work (Sec. 8) and conclude (Sec. 9), with lessons learned about dissemination of a research tool.

2 Initial Design of Coccinelle

Coccinelle development began in 2006. It was first made publicly available in binary in 2007 and in open source in 2008. We first review the original design decisions for Coccinelle, in terms of goals, expressivity, performance, correctness guarantees, and dissemination.

2.1 Goals

Coccinelle was initially designed to solve a specific problem, that of porting Linux device drivers from Linux 2.4, a previous stable version, to stable version Linux 2.6, which had been released shortly before the start of the project. The initial design was motivated by an earlier paper on *collateral evolutions* in the Linux kernel [45], *i.e.*, evolutions needed in API clients in response to changes in the API interface. The examples from that paper showed that to automate Linux kernel collateral evolutions it would be necessary to support transformations on scattered parts of the source code with various kinds of connections between them, including intraprocedural control-flow paths with specific properties. As a small research project could not encode the entire porting activity, these kinds of connections, derived from program-analysis concepts, would need to be expressed in a way that would be accessible to Linux driver developers, who could carry on the work. Targeting driver developers furthermore implied that Coccinelle would have to allow the user to reason about the code as it is shown to him, without simplification to an internal representation, and that it would have to treat a very large subset of C constructs, including various gcc extensions, according to the needs of arbitrary Linux kernel device driver code. Finally, the generated code would have to retain the structure of the original source code, including comments and whitespace, to ensure the code's continuing maintainability.

```
1 @ rule1 @
2 identifier fn, irq, dev_id;
3 typedef irqreturn_t;
4 @@
5 static irqreturn_t
6 fn(int irq, void *dev_id)
7 { ... }

8 @@
9 identifier rule1.fn;
10 expression E1, E2, E3;
11 @@
12  fn(E1, E2
13 -    ,E3
14     )
```

Figure 1: The first semantic patch submitted to Linux

2.2 Design decisions affecting expressivity

Coccinelle provides a transformation language SmPL (Semantic Patch Language) and an engine for applying SmPL semantic patches to C code. SmPL was conceived as a code pattern-matching language, mimicking the patch syntax. A SmPL semantic patch consists of a series of rules, analogous to patch hunks, each providing a code pattern to match or transform. Patterns are comprised of concrete syntax, "...", and metavariables. Concrete syntax matches itself, "..." matches a possibly empty sequence of arbitrary terms, *e.g.*, the list of statements between two other statements, and metavariables match arbitrary terms of a particular type. Metavariables are declared in a rule header and are used as ordinary variables in the pattern, to make the patterns close to the source code. Ideally, a Linux kernel developer should be able to copy a typical code example and add metavariable declarations, "...", and - and + annotations, to obtain a transformation rule with minimal effort.

The semantic patch in Figure 1 illustrates the various features of SmPL. This semantic patch completes an evolution and associated collateral evolutions that had been initiated by a Linux developer. The evolution changed the type of a callback function, by removing its third parameter. This required additionally removing the third argument from direct calls to this function. This change is challenging because the names of the affected functions are all different, implying that grep may not be sufficient to find all occurrences. A common strategy is to identify code to fix by compiler errors, iterating until the kernel compiles successfully. In this case, some of the direct calls had been overlooked due to being under ifdefs or in the support for obscure architectures.

The semantic patch consists of two rules, on lines 1-7 and lines 8-14, respectively. The first rule, named rule1, declares three identifier metavariables fn, irq and dev_id, representing the name of the function to match and the names of its two parameters, respectively. The rest of the first rule is a pattern that matches a function definition, in which the parameters and return value are indicated to have specific types (lines 5-6) and the body is allowed to be an arbitrary sequence of statements (line 7). The second rule, which has no name, declares four metavariables: the function name `fn`, whose value is explicitly inherited from the previous rule (line 9), and three expression metavariables, `E1`, `E2`, and `E3`, representing arbitrary argument expressions (line 10). The rest of this rule matches a call to the function that was identified in the first rule. In this call, the third argument is indicated to be removed (line 13). A wider variety of semantic patches is illustrated in various publications [31, 43, 50] and at the Coccinelle website [10, 11].

To apply a semantic patch to a code base, Coccinelle processes the C source code files one at a time. On each file, it applies the first rule of the semantic patch to each function or other top-level declaration, then applies the second rule to the code resulting from the first rule application, etc. Based on the needs observed in the prior collateral evolution study, the processing of a function is based on its intraprocedural control-flow graph. Thus, at the statement level, *e.g.*, line 7 of Figure 1, "..." follows intraprocedural control-flow paths, using a semantics based on a variant of CTL model checking [5]. By default, a pattern must match all control-flow paths starting from the control-flow graph node matching the beginning of the pattern, to ensure that the semantic patch describes a consistent view of the program behavior. For example, when a pattern such as `A(); ... B();` matches code including a conditional, `B();` must be reachable from `A();` via both branches of the conditional. Alternatively, a rule or an individual instance of "..." can be annotated with `exists` to indicate that only the existence of a matching path is required. By default, "..." cannot contain any code that is matched by the code pattern immediately preceding or following it, *e.g.*, to allow matching a call to a locking function and to the unlocking call closest to it, as needed due to the fine-grained locking found in the Linux kernel. Finally, a metavariable must match identical terms within a single control-flow path, but may match different terms in different control-flow paths, *e.g.*, different conditional branches [5].

2.3 Design decisions affecting performance

Coccinelle is intended to be used by a Linux developer in the course of his ordinary work, whenever a recurring transformation is needed. Accordingly, it must be usable on a standard laptop without much disruption. A number of the initial design decisions were guided by this goal.

The Linux kernel is very large, and indeed has more than tripled in size between Feb. 2007 (version 2.6.20, 5M LOC) and Jan. 2018 (version 4.15, 16.5M LOC). Processing the entire code base and achieving reasonable performance on a developer's laptop, thus requires making some tradeoffs. To reduce running time, Coccinelle focuses on regions of code that are most likely to be relevant for collateral evolutions, at the expense of the rest. A key observation is that an individual Linux kernel file typically addresses a problem at a given level of abstraction, while references to other files, via `#include` or function calls, typically move to a lower level of abstraction. Thus, the contents of header files and called functions may be less relevant for collateral evolutions.

Based on the above observation, by default, Coccinelle processes only `.c` files, includes only header files that are located in the same directory as the `.c` file or that have the same name as the `.c` file, and does not perform interprocedural analysis. Command-line options are provided to additionally process header files, independently of any files into which they may be included, and to include header files directly referenced in a `.c` file or all header files referenced recursively. The latter options, however, increase the amount of code processed, and thus the processing time. The use of these strategies is thus left up to the user, who is expected to know whether such information is relevant to the desired evolution. Finally, interprocedural analysis within a single file can be encoded up to a finite depth using a series of SmPL rules, each of which matches the definition of a function for which a function call was identified by a previous rule. More general interprocedural analysis originally required the use of external scripts to collect the names of called functions and to restart Coccinelle to process their definitions.

The only program analysis performed by Coccinelle is type inference. This analysis is best-effort, as non-inclusion of header files means that type information may be unavailable. Although the type information is incomplete, the inclusion of type information makes Coccinelle very useful for tasks such as finding where a field of a particular type of structure is referenced, without knowing the name of the variable pointing to that structure. Coccinelle performs no alias analysis or other form of dataflow analysis. Semantic patches that require control over aliases have to implement it explicitly, *e.g.*, by declaring that the code region matched by "..." cannot store the address of a given variable. This approach saves execution time, as the analysis is only performed if and to the extent that it is expected to be useful, and improves the predictability of the tool, as the semantic patch writer knows the strengths and limitations of the analysis.

2.4 Design decisions affecting correctness guarantees

Automatic program transformation has the potential to update code at a large scale reliably and efficiently, but it can also introduce pervasive bugs across a code base, if the transformation rules are incorrect or are implemented incorrectly by the transformation engine. Coccinelle only checks that a rule preserves the structural well formedness of the code, *e.g.*, ensuring that a statement is replaced by a statement, an expression by an expression, etc. It does not check for semantic correctness. This enables encoding bug fixes, which are intrinsically not semantics preserving. Furthermore, it enables efficiently applying rules, without complicated, typically interprocedural, analysis to show correctness. The goal of Coccinelle is to allow the user to express his knowledge about the software and the required changes, in terms of code fragments that resemble the affected code and that can be easily checked to conform to the user's intent.

2.5 Dissemination strategy

Reaching potential users is always a challenge for research projects. An open-source development context provides a diverse audience, which increases the chance that individual users will pick up new approaches, but makes it harder to impose a new approach on the entire developer base than in a monolithic industry setting. This is particularly the case of the Linux kernel developer community, which puts few restrictions on the tools used by developers to create and manage code.

To validate the utility of Coccinelle and to encourage its use by Linux developers, the Coccinelle developers took the strategy of showing by example. The first submitted patches (*e.g.*, 632155e65944 on June 1, 2007 [61]) exploited only the Coccinelle parser [42]. Indeed, as Coccinelle does not expand macros or reduce ifdefs, its parser can find errors that are overlooked in typical compile testing. The first submitted patch generated by a semantic patch was 0da2f0f164f0 (July 5, 2007). This patch was created using the semantic patch of Figure 1, and included the semantic patch in the commit log. It updated five files in the `net`, `atm`, and `usb` directories.

The next patch dd00cc486ab1 based on a semantic patch was submitted on July 6, 2007, changing a call to the memory allocator `kmalloc` followed by a `memset` to clear the memory into a single call to the function `kzalloc`, added in 2005. This patch affected 166 calls distributed over 146 files. The semantic patch in the log message received the comment "Cool!" from a developer,[1] but the patch ran afoul of the Linux kernel requirement that patches on different parts of the kernel be submitted separately to the relevant maintainers. Indeed, Coccinelle had shifted the burden from performing the change, which was now fully automated, to routing the individual changes to the proper maintainers, for which no automatic support was then available.

The first patches explicitly mentioning "Coccinelle" were submitted in December 2007, fixing various missing resource-release errors (76832d841643, etc.). These attempted to set a precedent for how the tool should be used, by including the URL of the tool, as well as the XML-like tags `<smpl>` and `</smpl>` around the semantic patch, to ease the tracking of the use of the tool. The first commits from outside the group of Coccinelle developers, 77bbadd5ea89 and 52fd8ca6ad41, came in July 2008, from the developer of the kernel-level memory checker `kmemcheck`. These patches corrected the type of a flag passed to various lock-related function calls. Coccinelle was released as open source in October 2008.

3 Evolutions in the Coccinelle Design

In retrospect, the design decisions presented in the previous section reflect a number of fundamental hypotheses about how to design a program transformation system that will be useful to and used by kernel developers:

Expressivity: Linux kernel developers will find it easy and convenient to describe needed code changes in terms of fragments of removed and added code.

Performance: Many interesting Linux kernel evolutions can be implemented reliably without incurring the cost of collecting and correlating information in multiple C files. Indeed, Linux kernel development relies on humans, who typically focus on one file at a time, and thus all relevant information should be directly apparent in a single C file.

Correctness: Proving correctness is not necessary because Linux kernel developers can easily incorporate their knowledge of kernel invariants into a semantic patch. Giving the developer control over the rules implies that the developer can control the rate of false positives, and can easily check for them in the results.

Dissemination: It is effective to show how a tool can be useful, rather than attempting to impose its use.

While Coccinelle still builds on these hypotheses, experience in using the tool and feedback from Linux kernel developers have provided lessons that have motivated various evolutions. We highlight those that have had the greatest impact on the use and usability of the tool.

3.1 Expressivity evolutions

Many changes, both simple and complex (*e.g.*, Figure 1), can be expressed purely in terms of code structure. Some kinds of changes, however, require more semantic information. Two evolutions that have greatly enhanced the expressivity of Coccinelle have been the introduction of *position variables* and *scripting rules*.

Position variables. A position variable is a Coccinelle metavariable that matches the position where a term occurs in a file. Position variables allow rematching the

[1] https://lkml.org/lkml/2007/7/7/98

```
 1 @ rule1 @                            16 @@
 2 expression t, f, e;                  17 expression rule1.t, rule1.f,
 3 position p1, p2;                     18   rule2.d;
 4 @@                                   19 position rule1.p1, rule1.p2,
 5 init_timer@p1(&t);                   20   rule2.p3;
 6 ... when != f = e                    21 @@
 7 t.function =@p2 f;                   22 (
 8                                      23 - init_timer@p1(&t);
 9 @ rule2 @                            24 + setup_timer(&t,f,d);
10 expression rule1.t, d, e;            25 |
11 position rule1.p1, p3;               26 - t.function =@p2 f;
12 @@                                   27 |
13 init_timer@p1(&t);                   28 - t.data =@p3 d;
14 ... when != d = e                    29 )
15 t.data =@p3 d;
```

Figure 2: init_timer conversion

```
 1 @r@                                  14 @@
 2 expression E; statement S;           15 expression E; statement S;
 3 position p1,p2;                      16 position r.p1;
 4 @@                                   17 @@
 5 if@p1 (E);                           18 if@p1 (E)
 6   S@p2                               19 - ;
 7                                      20 S
 8 @script:python@
 9 p1 << r.p1; p2 << r.p2;
10 @@
11 if (p1[0].col >= p2[0].col):
12   cocci.include_match(False)
```

Figure 3: Drop spurious semicolon after if header

same code in a later rule, as well as ensuring that a match in one rule is different than a match in an earlier rule.

Figure 2 illustrates the use of position variables to convert calls to `init_timer` to `setup_timer` when the `init_timer` call is followed by initializations of the timer `data` and `function` fields. The first two rules (lines 1-15) identify instances of the same `init_timer` call with two different properties (the `when` annotations in lines 6 and 14 indicate assignments that should not occur in the matched region). The last rule (lines 16-29) transforms `init_timer` calls that satisfy both properties. This rule includes a disjunction, such that all of the relevant code fragments can be transformed at once, wherever they occur, as once a transformation takes place, all previously bound position variables are invalidated.

Scripting language interface. The scripting language interface was initially motivated by the goal of using Coccinelle for bug finding [56]. While bugs that mainly depend on the code structure, such as use after free, could be found, the pattern-matching features of Coccinelle were not sufficient to detect bugs such as buffer overflows that require reasoning about variable values.

To allow reasoning about arbitrary information, support was added in 2008 for scripting-language rules. The first language supported was Python, which was expected to be familiar to Linux developers. Coccinelle is implemented in OCaml, and OCaml scripting was added in 2010, for the convenience of the Coccinelle developers. Scripts were originally designed to filter sets of metavariable bindings established by previous rules. Figure 3 shows an example, which drops a semicolon after an `if` header if the subsequent statement is indented, suggesting that the latter statement is intended to be the `if` branch. A script rule compares the indentation of the two statements (line 11) and discards metavariable binding environments (line 12) in which the conditional is aligned with or to the right of the subsequent statement.

Ultimately, the original motivation for scripting, *i.e.*, finding bugs such as buffer overflows, was not successful. The code patterns were small and generic, and the scripts implementing the required analyses were complex. Still, scripting has been a major leap forward for the expressiveness of Coccinelle, and new scripting functionalities have been added as new needs have emerged. Early on, libraries were added for generating formatted error messages. In 2009, `initialize` and `finalize` scripts were introduced to allow defining state global to the processing of all files, to facilitate the collection of statistics. In 2010, scripts became able to create new code fragments to be stored in metavariables and inherited by subsequent rules. In 2016, to improve performance and reduce semantic patch size, it became possible to add script code to metavariable declarations, to define predicates that would discard metavariable bindings early in the matching process. Finally, scripting enables *iteration*, which allows a semantic patch to submit new “jobs” to the Coccinelle engine, in order to perform analysis across multiple files.

3.2 Performance evolutions

While avoiding including header files reduces the volume of code to process, the Linux kernel remains a large and growing code base. Furthermore, parsing the code without relevant macro definitions from header files involves using heuristics, which can increase the parsing time. Thus, further optimizations were needed.

Indexing. An early observation was that performance could be improved by not parsing files that could not be matched by the semantic patch. Indeed, many semantic patches contain keywords such as the names of API functions that must be present for the semantic patch to match and that occur only a moderate number of times in the Linux kernel. Coccinelle initially used the Unix command `grep` to find the files containing these keywords, but this was still slow, given the large code size.

A second approach was to use `glimpse` [24] to prepare an index in advance, and then to only process the files indicated by the index. As the index is smaller than the kernel source code and is organized efficiently, the use of glimpse substantially improves performance, particularly for semantic patches that involve kernel API functions. Nevertheless, glimpse was originally only freely available to academic users, had to be manually installed, and creating an index on each kernel update is time-consuming. In 2010, this was complemented by support for id-utils, which is part of many Linux distributions and for which index creation is much faster. In 2013, the support for users who do not have an index available was rewritten to essentially reimplement the grep operation in OCaml, reducing system calls and better taking into account the specific needs of Coccinelle.

Parallelism. By default Coccinelle works on each .c file independently, and thus is ripe for parallelism. Nevertheless, when Coccinelle was first developed, there was no convenient support for parallelism in OCaml. Instead, the Coccinelle distribution included a shell script to launch multiple Coccinelle instances in parallel, each covering a different part of the code base. Users, however, were uneasy about using Coccinelle via an external script. Furthermore, processing different files can require very different amounts of time, and the lack of load balancing in this static solution meant that many cores could end up idle. Meanwhile, the Parmap OCaml parallelization library [14] became available, and between 2015 and 2017 increasing support was provided for parallelism, still at the .c file level, within Coccinelle itself.

Supporting finer grained parallelism, at the function level, was also considered. Initial experiments, however, suggested that the cost of passing around the state built up within the matching of a given file outweighed the benefits of parallelism. In contrast, Coccinelle treats each file independently, so the amount of state that needs to be passed between processes is minimal.

3.3 Correctness guarantee evolutions

Unlike the other cases, there have been no major evolutions in the view of transformation correctness. After having created over 450 semantic patches that have led to kernel patches, the Coccinelle developers have found that the original hypothesis that giving the developer control over the rules enables them to easily check the results is mostly sufficient. The few errors, *e.g.*, [53], have come from misunderstanding of kernel invariants that would require a prohibitively complex and time consuming semantic analysis to infer and check. Kernel maintainers have indeed concluded in some cases that it was the original code that was written in an error prone way [38].

3.4 Dissemination strategy evolutions

Showing the value and capabilities of Coccinelle by the example of submitted patches generated initial interest in the tool. As the expressivity of Coccinelle evolved to permit the specification of more complex changes, it became apparent that it would also be beneficial to more directly teach developers how to use Coccinelle, and to enable Coccinelle users to interact with each other.

Four workshops were organized on the use of Coccinelle and advertised on the Coccinelle mailing list, attracting industry participants. The Coccinelle developers also presented the tool and offered tutorials in a variety of developer conferences, including those targeting open-source enthusiasts (*e.g.*, FOSDEM) and those specifically targeting Linux kernel developers (*e.g.*, Linux Plumbers). These presentations focused on the user-visible aspects of Coccinelle, such as how to write semantic patches and what results could be achieved, rather than the details of the internal design of the system, which were presented in research venues [5].

The work on Coccinelle was also picked up by the Linux Weekly News (LWN), which is the standard reference for issues around the development of the Linux kernel and other open-source software. Tutorial articles on Coccinelle appeared in 2009 [25] and 2010 [52], authored not by the Coccinelle developers, but by well-known kernel developers. LWN has also reported on various talks about Coccinelle [12, 16].

4 Performance Evaluation

Coccinelle is intended to be used by a kernel developer as part of the normal development process, on a standard professional laptop. Accordingly, Coccinelle's performance should be acceptable in this setting. We illustrate the performance on a Lenovo Thinkpad T460s with two hyperthreaded 2.30GHz cores (Intel(R) Core(TM) i5-6200U CPU), a 3M cache, and 12G RAM. Our experiments focus on the 59 semantic patches found in the Linux 4.15 kernel, using the report mode, which is supported by all these semantic patches.[2] Times are based on a single run, with a timeout of 30 seconds per file. We use id-utils indexing. Figure 4 presents the elapsed time when running the semantic patches on the Thinkpad laptop, using both cores, with hyperthreading. The semantic patches are sorted in order of increasing running time.

For the Linux kernel, there is a precise performance point of reference that is familiar to the kernel developer; the time to perform a complete compile of the Linux kernel itself. The elapsed time for full kernel compilation on the Thinkpad laptop with 4 threads (hyperthreading) with make clean; make allyesconfig; make, is 54 minutes. Based on the results shown in Figure 4,

[2] The semantic patches and Coccinelle version used contain some performance improvements that will appear in Linux 4.18 and Coccinelle 1.0.7, respectively.

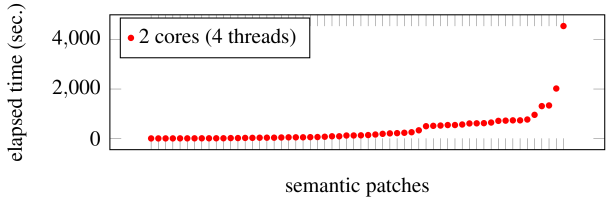


Figure 4: Elapsed time per semantic patch in the Linux 4.15 kernel on the Thinkpad laptop

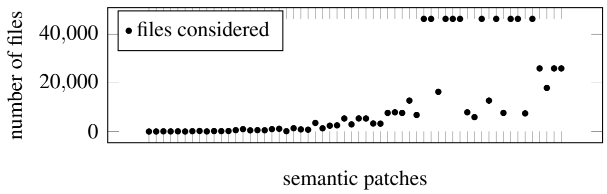


Figure 5: Files considered per Linux semantic patch.

all but one of the semantic patches complete within this time. The remaining semantic patch requires 75 minutes.

The timeout per file affects performance. In our experiment, Coccinelle timed out on 52 files, of 705,179 files considered, giving a timeout rate of 0.007%. Typically files on which timeouts occur contain complex code such as long sequences of loops and conditionals. These files can be analyzed separately, when more time is available.

Figure 5 shows the impact of indexing using id-utils on the number of files considered. The largest number of files considered is 46,336, in 10 cases, where no keywords are inferred from the semantic patch. These are associated with larger, but not the largest, execution times. At the far right of the graph, between 5000 and 26,000 files are considered, but the cost of tracing through all possible intraprocedural execution paths overwhelms the savings obtained by processing fewer files.

In terms of header files, 43 of the semantic patches specify that no include files should be considered. 11 specify to use the default (local and same-named files), and 1 specifies that all explicitly included should be taken into account. To assess the cost and benefit of including header files, we take the 44 semantic patches from the Linux kernel that complete in our test configuration in under 10 minutes and test them with options forcing the inclusion of no header files, the default, and all explicitly included header files. As compared to inclusion of no header files, the default increases the run time by up to 90% and the inclusion of all explicitly included header files increases the run time by up to 10x. The number of reports ranges from 1631 for no headers to 1691 for all headers, with most of the few differences on `.h` files.

The performance studied here is only relevant when scanning the entire kernel. When checking a single modified file, the time should rarely exceed a few seconds per semantic patch. Indexing may identify some semantic patches as irrelevant, reducing the execution time.

5 Expressivity Extension Evaluation

The position variable and scripting extensions increase the expressivity of SmPL, but add concepts that are not found in C code and thus are not already familiar to Linux developers. We thus assess the degree to which these features are used in practice. We note, however, that our only source of information about semantic patches is from those found in the Linux kernel and from those included in commit messages. This information may be incomplete, because developers can omit or simplify semantic patches in the commit message

All of the semantic patches found in the Linux kernel use positions and scripts in order to generate output in the report mode. 20 semantic patches were contributed by developers from outside the Coccinelle team. 3325 commits up through Linux 4.15 contain semantic patches in the commit message. Of these 586 (18%) contain position variables and 165 (5%) contain scripts. 43% of the latter commits come from outside the Coccinelle team.

6 Impact on Linux

Over the past 10 years, Coccinelle has been increasingly applied to the Linux kernel, by both Coccinelle developers and Linux kernel developers. As of Linux 4.15, over 6000 commits in the Linux kernel are based on the use of Coccinelle. In this section, we give an overview of the impact of Coccinelle on the Linux kernel. Graphs by subsystem reflect commits up through the release of Linux 4.15 (Jan. 2018). Graphs by year end with 2017.

6.1 Changed lines per subsystem

Figure 6 shows the number of lines removed and added by commits using Coccinelle in various kernel subsystems. The most affected is `drivers`, with 57,882 removed lines and 77,306 added lines, followed by `arch`, `fs`, `net`, `sound`, and `include`, all of which are affected by thousands of removed or added lines. The predominance of `drivers` is not surprising, given that `drivers` makes up 67% of the Linux 4.15 kernel source code. `drivers` has also been a target for other bug finding and code reliability tools [35, 37, 48, 51, 57].

Figure 7 compares the numbers of removed and added lines to the number of code lines (non-blank, non-comment, measured using SLOCCount [54]) found in Linux 4.15. The rate of Coccinelle-motivated changed

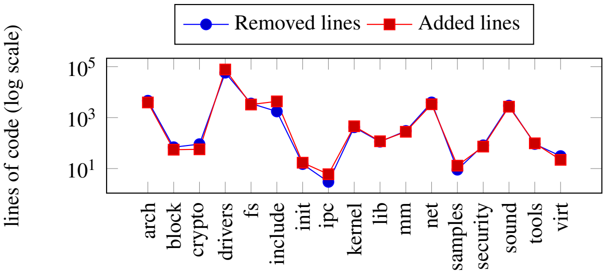


Figure 6: Number of lines removed and added by commits using Coccinelle, by subsystem

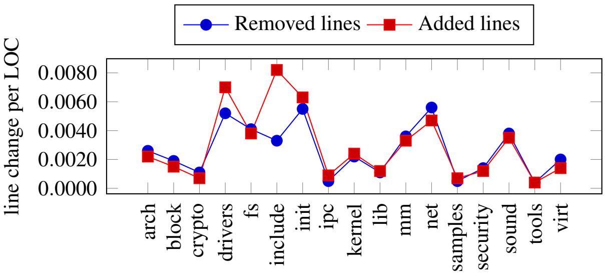


Figure 7: Lines removed and added by commits using Coccinelle per Linux 4.15 line of code, by subsystem

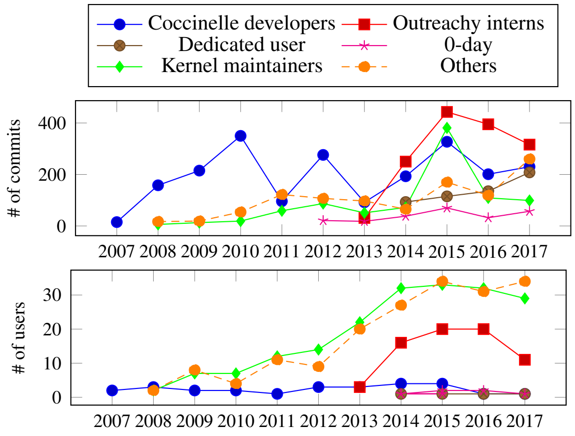


Figure 8: Number of commits using Coccinelle from various categories of Coccinelle users (top) and number of Coccinelle users in various categories having at least one commit using Coccinelle (bottom)

lines in drivers remains high, but the results show the applicability of Coccinelle across the kernel.

6.2 Categories of users over time

A variety of kinds of developers contribute to the Linux kernel, by submitting patches. Among those who mention Coccinelle in their commit logs, we distinguish six categories of Coccinelle users:

Coccinelle developers. These are members of the Coccinelle development team, and persons employed by the team to disseminate Coccinelle.

Outreachy interns. The Linux kernel participates in the Outreachy internship program [41] and interns may use Coccinelle in the application process or the internship.

Dedicated user. This is a single developer who uses Coccinelle in the kernel for a small collection of widely relevant simple changes.

0-day. This is an automated testing service at Intel that builds and boots the Linux kernel for multiple kernel configurations, on each commit to hundreds of git trees. The service also runs a number of static analysis tools, including Coccinelle, on the result of each commit.

Kernel maintainers. These are kernel developers who receive and commit patches, and are generally responsible for some subsystem's continued well being. We identify maintainers as developers who are named in the Linux 4.15 MAINTAINERS file (1170 developers) or who are the committer of some patch. Normally, kernel patches are transmitted by email, and only maintainers or developers the maintainers specifically designate commit to git trees that are pulled into a mainline release. Thus, being a committer is a sign of community respect.

Others. These are other Linux kernel contributors. These contributors may be frequent or occasional.

The top of Figure 8 shows the number of commits per year using Coccinelle from various kinds of Coccinelle users, while the bottom shows the number of Coccinelle users involved in each category. The first commits (2007) were from the Coccinelle development team. Use from maintainers and other kernel contributors started in the two years afterwards. The number of commits from maintainers has grown steadily, except for a major peak in 2015, when several maintainers undertook cross-tree refactoring projects using Coccinelle. The number of other kernel contributors has gone up and down, but shows an upward trend. These numbers may be underestimated, however, as some developers have revealed when asked that they used Coccinelle for repetitive changes, but did not mention it in the commit.[3]

The Linux developers who are most likely to have need for Coccinelle are those who perform large scale changes across the code base. To approximate this set of developers, we consider those who have at least one commit that touches at least 100 files, since Linux 3.0 (July 2011), *i.e.*, the period in which Coccinelle was becoming more established. There are 88 such developers, of which 67 (76%) are maintainers. All but two of the others are in the Other category. 39 (44%) of these develop-

[3] https://marc.info/?l=kernel-janitors&m=150403263119030&w=2

ers overall and 31 (46%) of these maintainers have commits using Coccinelle. Of the 88 developers, 27 (31%) have 1-5 commits using Coccinelle, 9 (10%) have 6-100 such commits, and 3 (3%) have more than 100. Among the 67 maintainers, 21 (31%) have 1-5 commits using Coccinelle, 8 (12%) have 6-100 such commits, and 2 (3%) have more than 100. These numbers suggest that Coccinelle is well known among the Linux kernel developers and maintainers who can benefit from it most.

Finally, we consider the most established kernel contributors. We collect the set of maintainers from the Linux 4.15 MAINTAINERS file and the set of developers who have committed at least one patch between Linux 3.0 and Linux 4.15. 45 have at least 10 years of experience as committers and 117 have committed at least 1000 patches. 29% and 32% of these, respectively, have created at least one patch that uses Coccinelle. These numbers reflect the knowledge of Coccinelle at the core of the Linux kernel developer community.

6.3 Changes performed using Coccinelle

Coccinelle facilitates performing changes across the kernel, that may cover code managed by multiple maintainers. Some examples are as follows:

TTY. Remove an unused function argument. One commit (429b474990cb) in 2015, affecting 11 TTY driver files. The author is not a maintainer, but has over 350 commits in the Linux kernel, since 2013.

IIO. Add missing `__devinit` and `__devexit` annotations. One commit (8e8287526844) in 2012 affecting 28 new IIO driver files. The author is an IIO maintainer.

DRM. Eliminate a redundant field in a data structure. One commit (438b74a5497c) in 2016 affecting 54 direct rendering manager (DRM) files. The author is a maintainer, but not for the affected files.

Interrupts. Prepare to remove the `irq` argument from interrupt handlers, and then remove that argument. 40 commits (*e.g.* f4acd122a738) in 2015, affecting 188 files (mostly `drivers`, `arch`). The author is a core Linux developer with over 3500 commits in the Linux kernel since the start of the Linux kernel's usage of git (2005).

More generally, Figure 9 characterizes as cleanups or bug fixes the complete set of patches that use Coccinelle from maintainers. Typical cleanups address generic C issues, such as useless double semicolons, as well as introductions of new APIs and refactorings in preparation for the introduction of new APIs. Commonly identified bugs include memory leaks, allocation of a memory region of the size of a pointer rather than the size of the referenced structure, and storing a potentially negative value in a variable of type `unsigned int` and then checking whether the value is less than zero.

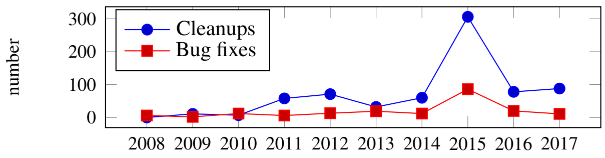


Figure 9: Cleanup vs. bug fix changes among maintainer patches using Coccinelle

As noted in Section 2.5, Coccinelle facilitates making changes that affect the entire Linux kernel source tree, and in particular subsystems managed by different maintainers. While the initial problem of knowing who maintains which part of the kernel was resolved in 2009 by the introduction of the `get_maintainer.pl` script, it is not always clear who should actually commit the changes, particularly when there are dependencies between the resulting patches. The problem of managing cross-tree changes was a proposed topic at the 2017 Linux Kernel Maintainers Summit. Preliminary discussions included proposing the use of Coccinelle for refreshing cross-tree changes when patch sets are incompletely applied [3].

6.4 Semantic patches in the Linux kernel

Since 2010, the Linux kernel has hosted a set of semantic patches in its `scripts/coccinelle` directory. Semantic patches are categorized as being related to APIs (17), resource release (7), iteration (5), locking (4), NULL values (4), test expressions (4) and others (18). The kernel Makefile contains a `coccicheck` target that runs one or all of these semantic patches on the entire kernel or some portion thereof. Kernel developers may thus easily use Coccinelle to check their work, without learning SmPL. Such use, however, is not visible in the kernel history.

As of Linux v4.15, there are 59 semantic patches in the Linux kernel. Figure 10 shows the number of commits including new semantic patches per year, as contributed by various categories of users. Semantic patches were initially contributed by the Coccinelle developers. Recently, 2-3 have been contributed each year, and a few more requested, by the wider kernel community.

6.5 0-day build testing service

Intel's 0-day testing service [26, 60] runs a number of static analyses on commits to hundreds of Linux kernel git trees, both public and private. Kernel developers may check their changes themselves on only one configuration and then rely on the 0-day service for the rest. Coccinelle-based reports generated by the 0-day build testing service come in two forms. If the semantic patch producing the report is able to propose a fix for the identified problem, then the report contains this patch. The

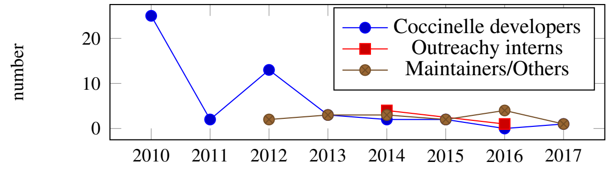


Figure 10: Number of commits adding a semantic patch to the Linux kernel source tree per year

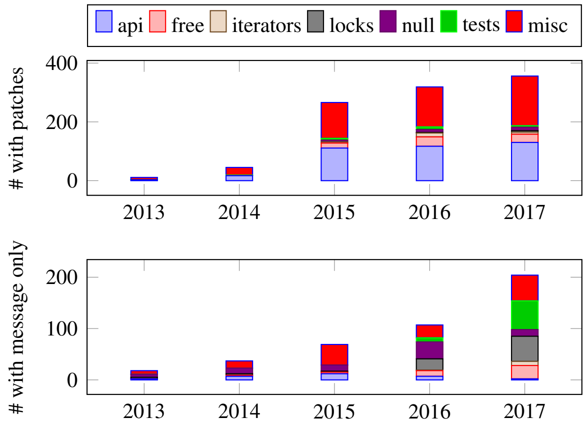


Figure 11: 0-day reports mentioning Coccinelle per year

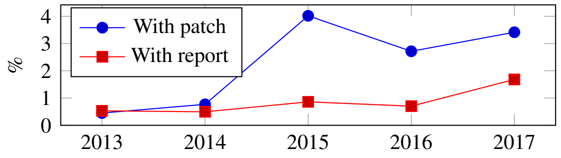


Figure 12: % of 0-day reports mentioning Coccinelle

remaining Coccinelle-based reports contain a textual error or warning message, accompanied by a code extract highlighting the relevant lines, as indicated by the semantic patch. Figure 11 shows the number of public reports that mention Coccinelle in various categories, distinguishing between those that include a patch or only a message. Figure 12 likewise shows the percentage of all public reports mentioning Coccinelle. The most common type of report including a patch removes a field initialization in drivers that is redundant with respect to the driver core (244 patches). The most common type of report including only a message detects missing unlocks (68 reports). The latter reports are manually checked by a Coccinelle developer and have few false positives. Both semantic patches involve kernel-specific features.

7 The Coccinelle Community

A measure of the long term potential impact of a project is the willingness and ability of external developers to contribute to the project's development and maintenance. Today Coccinelle amounts to over 84,000 lines of OCaml code. 25 developers have contributed to Coccinelle, with over 3000 commits over 12 years for one developer, almost 1000 in the first few years for another developer, and 200-300 commits in the last few years for several others. All of the contributors with more than 5 commits have been somehow affiliated with the core development team, as either an employee or a guest. These numbers are likely related to the fact that the implementation language of Coccinelle, OCaml, is not widely used in the target developer community, and to the interdisciplinary nature of Coccinelle, which builds on programming-language concepts but targets the systems developer community. The small number of contributions by external developers may be a source of long term fragility. Nevertheless, the fact that several developers have joined the project in recent years and each made around 200 or more commits suggests that the code is accessible to developers who did not initiate the project.

Coccinelle is packaged with a number of Linux distributions, such as Ubuntu [62], Debian [15], Fedora [18], Gentoo [23], and Archlinux [1]. It is also available for FreeBSD [21] and NetBSD [40]. The full commit history is available at Github [8]. Although Coccinelle is developed using OCaml, there has been an effort to limit the amount of dependence on the traditional OCaml culture and infrastructure. Some needed OCaml libraries are bundled with the Coccinelle distribution, in case they are not available on the local machine. Once Coccinelle is installed, it is fully usable, via the C-like SmPL language and Python scripting, without knowing OCaml.

Although Coccinelle is mainly used on the Linux kernel, it is also used on other software projects. RIOT [49], qemu [47] and systemd [58] include semantic patches in their source code distributions. Patches mentioning Coccinelle are also found in the commit histories of systems software projects such as cpython (d1302c01544 and 228b12edcce) [13], wine (f6ced24999f etc.) [64], and even one in Firefox (ab4e3a0d4213) [19]. The latter used Coccinelle's rudimentary support for C++.

8 Related Work

Academic software development tools. Other academic software development tools that have had an impact on industry practice include CIL [39], LLVM [29], and Metal [17]. CIL provides basic parsing and visitor infrastructure for processing C code, and is used for rapid prototyping as well as being at the base of mature tools such as Frama-C [20]. LLVM is a compiler infrastructure that originally targeted providing good support for static, link-time, and run-time optimization, and has evolved into a common alternative to gcc, due to its customizability, speed and space efficiency, and permissive license [7]. Neither has specifically targeted the Linux kernel and its particular coding style. Indeed, LLVM still does not fully support Linux kernel code, despite a long refactoring effort [33]. Both tools have furthermore focused on building a user community rather than taking on the challenge of integrating into an existing one.

Metal is an automata-based tool for scanning large systems source code bases for faults such as use after free and inconsistent locking. It was never made publicly available. Instead, it was the foundation of the highly successful static analysis tool Coverity [2]. Coverity has been used on the Linux kernel more or less over the years, depending on the degree to which its results have been made freely available. Nevertheless, the freely available results address generic C issues, rather than Linux specific properties.

Development tool impact analysis. Koyuncu *et al.* [28] compare properties of Linux kernel patches that are entirely manually generated, manually generated in response to a tool report, and tool generated according to a manually written transformation rule. The patches in the third category are primarily generated by Coccinelle. They find that manually generated patches are accepted more quickly than tool-supported patches, but that the acceptance rate of the latter is increasing. In contrast, we study what kinds of Linux kernel developers use Coccinelle, for what purpose, and what features of Coccinelle have led to its acceptance.

Other tools used on the Linux kernel. Checkpatch is a regular-expression based style checker, whose use is required by the Linux kernel patch submission checklist [32]. It does not have a global view of the code, so it cannot detect inconsistencies that involve multiple code fragments, such as a variable declaration and its use.

Sparse [4] was an effort by Linus Torvalds to develop an open source static checker for the Linux kernel, in response to Metal. Sparse processes developer-provided annotations, enabling it to, *e.g.*, detect endianness issues. Smatch [55] grew out of sparse as a more flexible bug finding tool. Like Coccinelle, Smatch is scriptable, but rules are expressed at the abstract-syntax tree level rather than at the source code level. Thus, the user needs to know about internal representations. Smatch also does only bug finding, not transformation. On the other hand, Smatch tracks variable values, while Coccinelle reasons only in terms of code structure. Thus, smatch can find bugs such as off-by-one errors that are difficult to find using Coccinelle. Thousands of commits in the Linux kernel are derived from the use of smatch, but the creation of new rules is mostly limited to the tool author.

Another academic effort on improving Linux kernel code is the Linux Driver Verification project [27]. It centers around developing infrastructure and rule sets making it possible to apply verification tools such as CPAchecker and BLAST to the Linux kernel. A few hundred commits in the Linux kernel are based on its results. The Undertaker project [6, 59], in contrast to the other tools, finds bug in the use of configuration variables.

9 Lessons Learned

In this paper, we have reviewed the evolution of the program transformation tool Coccinelle and its impact on the Linux kernel. The experience of Coccinelle can help guide other projects that want to have an impact on an open source systems developer community.

First, visibility is necessary. The Coccinelle developers taught by example, by using Coccinelle to make a sustained contribution to the Linux kernel. At the same time, they organized workshops on the use of Coccinelle and presented Coccinelle in a variety of developer conferences, both focusing on the user-visible aspects of Coccinelle, to make the tool accessible to developers.

Second, the tool must be easy to install and freely available. Coccinelle is implemented in OCaml, but targets C developers. There has been a concerted effort to minimize reliance on OCaml infrastructure. The cost is a complex build system, but it reduces the chance that users will immediately abandon Coccinelle because it is difficult to install. Likewise, Coccinelle is freely available (GPLv2), with no registration requirement.

Third, the tool must be easy to use and robust, with support to quickly address problems encountered by users. While many research prototypes are only robust enough to complete an evaluation for a paper submission, users will try it on all kinds of code, and use it in unanticipated ways. A strength of Coccinelle is its lax C parser, motivated by the need to parse code without reliance on header files. This has the side effect of allowing it to adapt to C variants used by different projects. On the other hand, many users have mentioned that the documentation, consisting mainly of a BNF, some examples and some tutorial presentations, is hard to understand. Nevertheless, Coccinelle has an active mailing list [9] on which user problems are quickly addressed. Fixes are made available quickly via Github [8].

Finally, in a research setting, there is a constant temptation to make a tool do more, until the resulting complexity causes the tool to collapse under its own weight. While new features have been added to Coccinelle, the tool has remained within the scope of pattern matching-based transformation of C code. This focus has allowed it to grow and achieve practical success in this area.

Availability. `http://coccinelle.lip6.fr`. This work was supported in part by ANR-NRF ITrans.

References

[1] Archlinux. https://aur.archlinux.org/packages/coccinelle.

[2] BESSEY, A., BLOCK, K., CHELF, B., CHOU, A., FULTON, B., HALLEM, S., GROS, C.-H., KAMSKY, A., MCPEAK, S., AND ENGLER, D. R. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (2010), 66–75.

[3] BOTTOMLEY, J. Maintainer's summit agenda planning. https://lists.linuxfoundation.org/pipermail/ksummit-discuss/2017-October/004803.html.

[4] BROWN, N. Sparse: a look under the hood, June 2016. https://lwn.net/Articles/689907.

[5] BRUNEL, J., DOLIGEZ, D., HANSEN, R. R., LAWALL, J. L., AND MULLER, G. A foundation for flow-based program matching: using temporal logic and model checking. In *POPL* (Savannah, GA, USA, 2009), pp. 114–126.

[6] Checkkconfigsymbols. https://github.com/torvalds/linux/blob/master/scripts/checkkconfigsymbols.py.

[7] Clang vs other open source compilers. http://clang.llvm.org/comparison.html.

[8] Coccinelle github repository. https://github.com/coccinelle/coccinelle.

[9] Coccinelle mailing list. https://systeme.lip6.fr/pipermail/cocci.

[10] Coccinelle website. http://coccinelle.lip6.fr/.

[11] Coccinellery. https://github.com/coccinelle/coccinellery.

[12] CORBET, J. KS2010: Lightning talks, Nov. 2010. https://lwn.net/Articles/412750.

[13] CPython. https://github.com/python/cpython.git.

[14] DANELUTTO, M., AND COSMO, R. D. A "minimal disruption" skeleton experiment: seamless map & reduce embedding in OCaml. In *International Conference on Computational Science* (Omaha, NE, USA, June 2012), pp. 1837–1846.

[15] Debian. https://packages.debian.org/search?keywords=coccinelle.

[16] EDGE, J. Inside the mind of a Coccinelle programmer, Aug. 2016. https://lwn.net/Articles/698724.

[17] ENGLER, D. R., CHELF, B., CHOU, A., AND HALLEM, S. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI* (2000), pp. 1–16.

[18] Fedora. https://apps.fedoraproject.org/packages/coccinelle.

[19] Firefox. https://github.com/mozilla/gecko-dev.git.

[20] Frama-C. https://frama-c.com.

[21] FreeBSD. http://www.freshports.org/devel/coccinelle.

[22] Gartner says worldwide sales of smartphones grew 9 percent in first quarter of 2017. https://www.gartner.com/newsroom/id/3725117.

[23] Gentoo. https://packages.gentoo.org/packages/dev-util/coccinelle.

[24] Glimpse. http://webglimpse.net.

[25] HENSON, V. Semantic patching with Coccinelle, Jan. 2009. https://lwn.net/Articles/315686.

[26] KERRISK, M. Ks2012: Kernel build/boot testing, Sept. 2012. https://lwn.net/Articles/514278.

[27] KHOROSHILOV, A., MANDRYKIN, M., MUTILIN, V., NOVIKOV, E., PETRENKO, A., AND ZAKHAROV, I. Configurable toolset for static verification of operating systems kernel modules. *Programming and Computer Software 41*, 1 (2015), 49–64.

[28] KOYUNCU, A., BISSYANDÉ, T. F., KIM, D., KLEIN, J., MONPERRUS, M., AND TRAON, Y. L. Impact of tool support in patch construction. In *ISSTA* (2017).

[29] LATTNER, C., AND ADVE, V. S. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO* (2004), pp. 75–88.

[30] LAWALL, J., PALINSKI, D., GNIRKE, L., AND MULLER, G. Fast and precise retrieval of forward and back porting information for Linux device drivers. In *USENIX Annual Technical Conference* (2017), pp. 15–26.

[31] LAWALL, J. L., BRUNEL, J., PALIX, N., HANSEN, R. R., STUART, H., AND MULLER, G. WYSIWIB: exploiting fine-grained program structure in a scriptable API-usage protocol-finding process. *Softw., Pract. Exper. 1*, 43 (2013), 67–92.

[32] Linux kernel patch submission checklist. https://www.kernel.org/doc/html/v4.15/process/submit-checklist.html.

[33] LLVM. http://llvm.linuxfoundation.org/index.php/Main_Page.

[34] LOZI, J.-P., DAVID, F., THOMAS, G., LAWALL, J. L., AND MULLER, G. Fast and portable locking for multicore architectures. *ACM Trans. Comput. Syst. 4*, 33 (2016), 13:1–13:62.

[35] MACHIRY, A., SPENSKY, C., CORINA, J., STEPHENS, N., KRUEGEL, C., AND VIGNA, G. DR. CHECKER: A soundy analysis for Linux kernel drivers. In *USENIX Security* (Vancouver, BC, Canada, 2017).

[36] MACKENZIE, D., EGGERT, P., AND STALLMAN, R. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd, Jan. 2003. Unified Format section, http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html.

[37] MÉRILLON, F., RÉVEILLÈRE, L., CONSEL, C., MARLET, R., AND MULLER, G. Devil: An IDL for hardware programming. In *OSDI* (San Diego, CA, USA, 2000), USENIX Association.

[38] MOLNAR, I. Revert "make 'bt_sfi_data' const". https://lkml.org/lkml/2017/12/28/137.

[39] NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction, 11th International Conference* (Grenoble, France, Apr. 2002), LNCS 2304, pp. 213–228.

[40] NetBSD. ftp://ftp.netbsd.org/pub/pkgsrc/current/pkgsrc/devel/coccinelle/README.html.

[41] Outreachy. https://www.outreachy.org.

[42] PADIOLEAU, Y. Parsing C/C++ code without pre-processing. In *CC* (York, UK, Mar. 2009), pp. 109–125.

[43] PADIOLEAU, Y., LAWALL, J., HANSEN, R. R., AND MULLER, G. Semantic patches for collateral evolutions in device drivers. In *Linux Symposium* (Ottawa, Canada, June 2007).

[44] PADIOLEAU, Y., LAWALL, J. L., HANSEN, R. R., AND MULLER, G. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys* (2008), pp. 247–260.

[45] PADIOLEAU, Y., LAWALL, J. L., AND MULLER, G. Understanding collateral evolution in Linux device drivers. In *EuroSys* (2006), pp. 59–71.

[46] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., MULLER, G., AND LAWALL, J. Faults in Linux 2.6. *ACM Trans. Comput. Syst. 32*, 2 (2014), 4:1–4:40.

[47] Qemu. `https://github.com/qemu/qemu.git`.

[48] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. Symdrive: Testing drivers without devices. In *OSDI* (Hollywood, CA, 2012), USENIX, pp. 279–292.

[49] Riot. `https://github.com/RIOT-OS/RIOT.git`.

[50] RODRIGUEZ, L. R., AND LAWALL, J. Increasing automation in the backporting of Linux drivers using Coccinelle. In *EDCC* (2015), pp. 132–143.

[51] RYZHYK, L., KEYS, J., MIRLA, B., RAGHUNATH, A., VIJ, M., AND HEISER, G. Improved device driver reliability through hardware verification reuse. In *ASPLOS* (2011), pp. 133–144.

[52] SANG, W. Evolutionary development of a semantic patch using Coccinelle, Mar. 2010. `https://lwn.net/Articles/380835`.

[53] SHEVCHENKO, A. Revert "make 'bt_sfi_data' const". `https://lkml.org/lkml/2017/12/28/85`.

[54] SLOCCount. `https://www.dwheeler.com/sloccount`.

[55] Smatch. `https://blogs.oracle.com/linuxkernel/smatch-static-analysis-tool-overview,-by-dan-carpenter`.

[56] STUART, H., HANSEN, R. R., LAWALL, J. L., ANDERSEN, J., PADIOLEAU, Y., AND MULLER, G. Towards easing the diagnosis of bugs in OS code. In *4th Workshop on Programming Languages and Operating Systems* (Stevenson, Washington, Oct. 2007).

[57] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering device drivers. *ACM Trans. Comput. Syst. 24*, 4 (Nov. 2006), 333–360.

[58] Systemd. `https://github.com/systemd/systemd`.

[59] TARTLER, R., SINCERO, J., DIETRICH, C., SCHRÖDER-PREIKSCHAT, W., AND LOHMANN, D. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer 14*, 5 (2012), 531–551.

[60] The kbuild-all archives. `https://lists.01.org/pipermail/kbuild-all`.

[61] TORVALDS, L. Linux kernel source tree. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/`.

[62] Ubuntu. `https://packages.ubuntu.com/zesty/coccinelle`.

[63] VAUGHAN-NICHOLS, S. J. Linux totally dominates supercomputers. `http://www.zdnet.com/article/linux-totally-dominates-supercomputers`.

[64] Wine. `https://github.com/wine-mirror/wine.git`.

Albis: High-Performance File Format for Big Data Systems

Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler
IBM Research, Zurich

Abstract

Over the last decade, a variety of external file formats such as Parquet, ORC, Arrow, etc., have been developed to store large volumes of relational data in the cloud. As high-performance networking and storage devices are used pervasively to process this data in frameworks like Spark and Hadoop, we observe that none of the popular file formats are capable of delivering data access rates close to the hardware. Our analysis suggests that multiple antiquated notions about the nature of I/O in a distributed setting, and the preference for the "storage efficiency" over performance is the key reason for this gap.

In this paper we present Albis, a high-performance file format for storing relational data on modern hardware. Albis is built upon two key principles: (i) reduce the CPU cost by keeping the data/metadata storage format simple; (ii) use a binary API for an efficient object management to avoid unnecessary object materialization. In our evaluation, we demonstrate that in micro-benchmarks Albis delivers $1.9 - 21.4\times$ faster bandwidths than other formats. At the workload-level, Albis in Spark/SQL reduces the runtimes of TPC-DS queries up to a margin of $3\times$.

1 Introduction

Relational data management and analysis is one of the most popular data processing paradigms. Over the last decade, many distributed relational data *processing* systems (RDPS) have been proposed [15, 53, 29, 38, 35, 24]. These systems routinely process vast quantities of (semi-)structured relational data to generate valuable insights [33]. As the volume and velocity of the data increase, these systems are under constant pressure to deliver ever higher performance. One key factor that determines the performance is the data access rate. However, unlike the classic relational database management systems (RDBMS) which are jointly designed for optimal data storage and processing, modern cloud-based RDPS systems typically do not manage their storage. They leverage a variety of external file formats to store and access data. Figure 1 shows a typical RDPS stack in the cloud. This modularity enables RDPS systems to access data from a variety of sources in a diverse set of deployments. Examples of these formats are Parquet [10], ORC [9], Avro [6], Arrow [5], etc. These formats are now even supported by the RDBMS solutions which add Hadoop support [49, 41, 31]. Inevitably, the performance of a file format plays an important role.

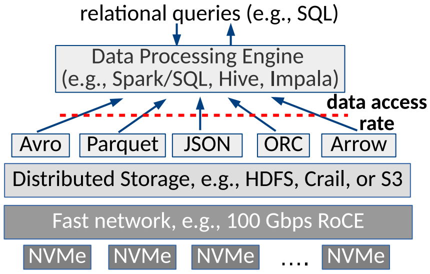

relational queries (e.g., SQL)
Data Processing Engine
(e.g., Spark/SQL, Hive, Impala)
data access rate
Avro
Parquet
JSON
ORC
Arrow
Distributed Storage, e.g., HDFS, Crail, or S3
Fast network, e.g., 100 Gbps RoCE
NVMe
NVMe
NVMe
....
NVMe

Figure 1: Relational data processing stack in the cloud.

Historically, file formats have put the top priority as the "storage efficiency", and aim to reduce the amount of I/O as much as possible because I/O operations are considered slow. However, with the recent performance advancements in storage and network devices, the fundamental notion of *"a fast CPU and slow I/O devices"* is now antiquated [44, 40, 54]. Consequently, many assumptions about the nature of storage in a distributed setting are in need of revision (see Table 1). Yet, file formats continue to build upon these antiquated assumptions without a systematic consideration for the performance. As a result, only a fraction of raw hardware performance is reflected in the performance of a file format.

In this paper, we re-visit the basic question of storage and file formats for modern, cloud-scale relational data processing systems. We first start by quantifying the impact of modern networking and storage hardware

Assumption	Implications	Still valid in a modern setup?
1. I/O operations are orders of magnitude slower than the CPU	Use compression and encoding to reduce the amount of I/O required	No, with high-performance devices, the CPU is the new performance bottleneck [54, 20]
2. Random, small I/O accesses are slow	Use large block sizes to make large sequential scans [30, 29]	No, modern NVMe devices have high-performance for random, small accesses
3. Avoid remote data access	Preserve locality by packing data in discrete storage blocks [25]	No, fast I/O devices with network protocols (e.g., NVMeF) make remote storage as fast as local [26, 34]
4. Metadata lookups are slow	Decrease the number of files and blocks needed to store data [30]	No, high-performance distributed storage systems can do millions of lookups per second [50]
5. The final row representation is not known	Provide object/data-oriented API for row and column data types	No, the final row/column representation is often known (e.g., Spark `UnsafeRow`) and a binary API can be used

Table 1: Assumptions (1–2 are local, and 3–5 are distributed) and their implications on storing relational data.

on the performance of file formats. Our experiments lead to three key findings. First, no popular file format we test can deliver data access bandwidths close to what is possible on modern hardware. On our 100 Gbps setup, the best performer delivers about ⅓rd of the bandwidth of the networked storage. Secondly, the CPU is the new bottleneck in the data access pipeline. Even in the presence of high-performance I/O hardware, file format developers continue to trade the CPU performance for "efficient" I/O patterns. Although this decision made sense for disks and 1 Gbps networks, today, this leads to the CPU being kept busy with (de)compressing, en/decoding, copying data, managing objects, etc., while trying to keep up with incoming data rates. Lastly, at the distributed systems level, strict adherence to locality, preference for large sequential scans, penchant to decrease the number of files/blocks, and poor object management in a managed run-time result in a complex implementation with a very high CPU cost and a poor "COST" score [37].

Based upon these findings, in this paper, we propose Albis, a simple, high-performance file format for RDPS systems. Albis is developed upon two key principles: (i) reduce the CPU cost by keeping the data/metadata storage format simple; (ii) use a binary API for an efficient object management to avoid unnecessary object materialization. These principles then also simplify the data/file management in a distributed environment where Albis stores schema, metadata, and data in separate files for an easy evolution, and does not enforce a store like HDFS to use local blocks. In essence, Albis's top priority is to deliver performance of the storage and network hardware without too much intrusion from the software layer. Our specific contributions in this paper are:

- Quantification of the performance of popular file formats on modern hardware. To the best of our knowledge, this is the first systematic performance evaluation of file formats on 100 Gbps network and NVMe devices. Often, such evaluations are muddied in the description of the accompanying relational processing system, which makes it hard to understand where the performance bottlenecks in the system are.
- Revision of the long held CPU-I/O performance assumptions in a distributed setting. Based upon these revisions, we propose Albis, a high-performance file format for relational data storage systems.
- Evaluation of Albis on modern hardware where we demonstrate that it can deliver performance within 15% (85.5 Gbps) of the hardware. Beyond microbenchmarks, we also integrate Albis in Spark/SQL and demonstrate its effectiveness with TPC-DS workload acceleration where queries observe gains up to 3×.

2 File Formats in the Age of Fast I/O

The choice of a file format dictates how multi-dimensional relational tables are stored in flat, one-dimensional files. The initial influential work of column-oriented databases have demonstrated the effectivness of column storage for disks [51, 12]. This has led to the development of a series of columnar file formats. The most prominent of them are Apache Parquet [10], Optimized Row Columnar (ORC) [9], and Arrow [5]. All of these columnar formats differ in their column/storage encoding, storage efficiency, and granularity of indexing. Apart from providing a row-at-a-time API, all of these formats also have a high-performance vector API for column data. In this paper, we use the vector API for evaluation. In contrast to column-storage, we also consider two popular row-oriented formats. JSON [11] is a simple data representation that encodes schema and data together. JSON is widely supported due to its simplicity. The other format is Avro [6] that decouples schema and data presentation where both can evolve independently.

2.1 The Mismatch Assumptions

In this section, we re-visit the basic assumptions made about the nature of I/O and what impact they have on the

	1 GbE	100 GbE	Disk	NVMe
Bandwidth	117 MB/s	12.5 GB/s	140 MB/s	3.1 GB/s
`cycles/unit`	38,400	360	10,957	495

Table 2: Bandwidths and `cycles/unit` margins for networking and storage devices. A unit for network is a 1,500 bytes packet, whereas for storage it is a 512 bytes sector. `Cycles/unit` roughly denote the number of free CPU cycles for every data unit for a 3 GHz core. As a reference, a DRAM-access would be around 100 cycles.

file format design in the presence of modern hardware. This discussion is summarized in Table 1.

1. I/O operations are orders of magnitude slower than the CPU: During the last decade, we have witnessed the rise of high-performance storage and networking devices like 40 and 100 Gbps Ethernet, and NVMe storage. Once the staple of high-performance computing clusters, these devices and associated APIs can now be found in commodity cloud offerings from multiple vendors [3, 2, 1]. At the same time, the CPU performance improvements have stalled due to various thermal and manufacturing limits. Hence, the CPU's margin for processing incoming bytes has shrunk considerably [45, 21, 16, 54]. In Table 2 we summarize the bandwidths for state-of-the-art I/O devices from a decade ago and now. We also show the `cycles/unit` metric as an estimate of the CPU budget for every incoming unit of data. For the network, the unit is a 1,500 bytes packet, and for storage it is a 512 bytes sector. For a 3 GHz core (ignoring the micro-architectural artifacts), the number of cycles per second is around 3×10^9. The table shows that in comparison to a decade ago, CPU cycle margins have shrunk by two orders of magnitude.

2. Random, small I/O accesses are slow: Disk seeks are slow and take $\sim$10s of milliseconds, a cost that cannot be amortized easily for small accesses. Hence, disk-based file formats advocate using large I/O segments, typically a multiple of the underlying storage block, e.g., 128 or 256 MB [30]. However, NVMe devices can deliver high bandwidth for random, small I/O patterns. In our investigation (discussed in the next section), we find that the continuing use of large I/O buffers is detrimental to the cache behavior and performance. For example, on a 16 core machine with a 128 MB buffer for each task, the memory footprint of a workload would be 2 GB, a much larger quantity than the modern cache sizes.

3. Avoid remote data access: Modern NVMe devices can do 2-3 GB/s reads and 1-2 GB/s writes. At the same time, the availability of high-performance networks (40 and 100 Gbps) and efficient network protocols like NMVe-over-Fabrics, means that the performance gap between a local flash and remote flash is negligible [26, 34]. Hence, various proposed modifications to block placement strategies in Hadoop [25], and design decisions to pack schema, data, and metadata in the same block to avoid remote storage, can be relaxed.

4. Metadata lookups are slow: In any distributed storage, the number of metadata lookups is directly proportional to the number of blocks in a file. A lookup is an RPC that took 100-1,000 μs over 1 Gbps networks. This high cost has led to the decision to reduce the number of files (or blocks) by using complex block management strategies, type-specific encoding, packing data and schema in packed blocks, which in essence trades CPU for the I/O. However, modern storage solutions like Apache Crail [8] and RAMCloud [42] can do millions of metadata lookups/sec [50].

5. The final data representation is not known: File formats often assume that the final data representation in an RDPS engine is not known, and hence, a format must materialize the raw objects when reading and writing data. This decision leads to unnecessary serialization and object allocation, which hampers the performance in a managed run-time environment like Java.

2.2 Putting it Together: Performance

In this section, we quantify the cumulative effect of the aforementioned assumptions on the read performance of file formats on modern hardware. For this experiment, we read and materialize values from the `store_sales` table (the largest table) from the TPC-DS (scale=100) dataset. The table contains 23 columns, which consist of 10 integers, 1 long, and 12 decimal values. The input table is stored in the HDFS file system (`v2.7`) in Parquet (`v1.8`), ORC (`v1.4`), Arrow (`v0.8`), Avro (`v1.7`), and JSON formats. The goal of the experiment is to measure how fast we can materialize the values from a remote storage. The experiment is run between 2 machines (with dual Xeon E5-2690, 16 cores) connected via a 100 Gbps network. One machine is used to run the HDFS namenode and a datanode. This machine also contains 4 enterprise-grade NVMe cards, with a cumulative bandwidth of 12.4 GB/sec. The other machine runs the benchmarking code[1] on all 16 cores in parallel.

Figure 2 shows our findings. Here, the y-axis shows the effective *goodput* calculated by dividing the total incoming data size by the runtime. Notice that the incoming *data size* is different from the *file size*, which depends upon the file format used. We cannot use the file size for the bandwidth calculation because formats such as JSON use text encoding with interleaved schemas, thus making their file sizes up to 10× larger than the actual data size. In order to measure the actual data content, we count how

[1] The benchmarking code is open-sourced at `https://github.com/animeshtrivedi/fileformat-benchmarks`.

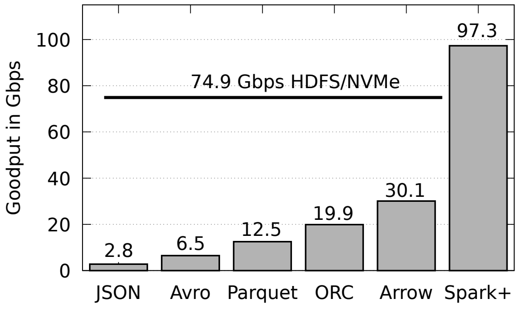


Figure 2: Reading bandwidths for various file formats.

		512M	256M	128M	64M	32M
Goodput (in Gbps)	Parq.	7.3	9.5	12.5	12.8	12.1
	ORC	13.6	17.5	19.9	20.2	20.1
Instructions/ row	Parq.	6.6K	6.7K	6.6K	6.6K	6.6K
	ORC	5.0K	4.9K	4.9K	4.8K	4.8K
Cache misses/ row	Parq.	11.0	10.6	9.2	7.1	6.5
	ORC	7.8	5.5	4.6	4.4	4.1

Table 3: Goodputs, instructions/row, and cache misses/row for Parquet (Parq.) and ORC with varying block sizes on a 16-core Xeon machine.

many integers, longs, and doubles the table contains, excluding the null values. We use this as the reference data point for the goodput measurements. We also show the data ingestion bandwidth of Spark/SQL [15] (as `spark+`) when the data is generated on the fly. The solid line represents the HDFS read bandwidth (74.9 Gbps) from the remote NVMe devices[2].

There are three key results from our experiment. First, as shown in Figure 2, none of the file formats that we test delivers bandwidth close to the storage performance. There is almost an order of magnitude performance gap between the HDFS (74.9 Gbps) and JSON (2.8 Gbps) and Avro (6.5 Gbps) performances. Columnar formats with their optimized vector API perform better. The best performer is Arrow, which delivers 40.2% of the HDFS bandwidth. Interestingly, Arrow is not for disks, but for in-memory columnar data presentation. Its performance only supports our case that with modern storage and networking hardware, file formats need to take a more "In-Memory"-ish approach to storage. In the same figure, we also show that in isolation from the storage/file formats, Spark/SQL can materialize `store_sales` rows from raw integers, longs, and doubles at the rate of 97.3 Gbps. Hence, we conclude that file formats are a major performance bottleneck for accessing data at high rates.

Secondly, the performance of these file formats are CPU limited, an observation also made by others [20]. When reading the data, all 16 cores are 100% occupied executing the thick software stack that consists of kernel code, HDFS-client code, data copies, encoding, (de)serialization, and object management routines. In Table 3 we present further breakdown of the performance (1^{st} row) with required instructions per row (2^{nd} row) and cache misses per row (3^{rd} row) for Parquet and ORC file formats when varying their block sizes. As shown, the use of large blocks (256 MB and 512 MB) always leads to poor performance. The key reason for the performance loss is the increased number of cache misses that leads to stalled CPU cycles. As we decrease the block size from 512 to 32 MB, the performance increases up to 128 MB, though the number of cache misses continues to decrease. At smaller block sizes (128 – 32 MB), the performance does not further improve because it is bottlenecked by the large number of instructions/row (remains almost constant as shown in the 2^{nd} row) that a CPU needs to execute. In further experiments, we use a 128 MB block size as recommended in the literature.

Thirdly, these inefficiencies are scattered throughout the software stack of file formats, and require a fresh and holistic approach in order to be fixed.

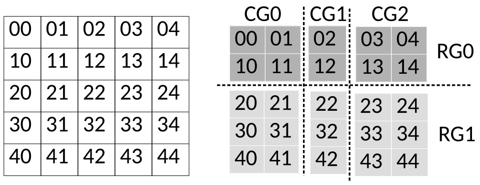
Summary: We have demonstrated that despite orders of magnitude performance improvements in networked-storage performance, modern file formats fail to deliver this performance to data processing systems. The key reason for this inefficiency is the belief in the legacy assumptions where CPU cycles are still traded off for I/O performance, which is not necessary anymore. Having shown the motivation for our choices, in the next section we present Albis, a high-performance file format.

3 Design of Albis File Format

Albis is a file format to store and process relational tabular data for read-heavy analytic workloads in a distributed setting. It supports all the primitive fixed (`int, timestamp, decimal, double`, etc.) and variable (`varchar` or `byte[]`) data types with simple and nested schemas. A nested schema is flattened over the column names and data is stored in the schema leaves. Albis's design is based upon the following choices:

- **No compression or encoding:** Albis decreases the CPU pressure by storing data in simple binary blobs without any encoding or compression. This decision trades storage space for better performance.
- **Remove unnecessary object materialization by providing a binary API:** The data remains in the binary format unless explicitly called for materialization. A

[2] Even though this is not 100 Gbps, it is the same bandwidth that HDFS can serve locally from NVMe devices. Hence, the assumption about the equality of local and remote performance holds.



/table1/schema
/table1/rg0/cg0/{data,mdata}.ab → [00,01]:[10,11]
/table1/rg0/cg1/{data,mdata}.ab → [02]:[12]
/table1/rg0/cg2/{data,mdata}.ab → [03,04]:[13,14]
/table1/rg1/cg0/{data,mdata}.ab → [20,21]:[30,31]:[40,41]
/table1/rg1/cg1/{data,mdata}.ab → [22]:[32]:[42]
/table1/rg1/cg2/{data,mdata}.ab → [23,24]:[33,34]:[43,44]

Figure 3: Table partitioning logic of Albis and corresponding file and directory layout on the file system.

reader can access the binary data blob for data transformation from Albis to another format (such as Spark's `UnsafeRow` representation) without materializing the data. This design choice helps to reduce the number of objects. A binary API is possible because the row data is not group compressed or encoded which requires the complete group to be decoded to materialize values.

- **Keep the metadata/data management simple:** Albis stores schema, data, and metadata in separate files with certain conventions on file and directory names. This setup helps to avoid complex inter-weaving of data, schema, and metadata found in other file formats. Due to the simple data/metadata management logic, Albis's I/O path is light-weight and fast.

In order to distribute storage and processing, Albis partitions a table horizontally and vertically as shown in Figure 3. The vertical partitioning (configurable) splits columns into *column-groups (CGs)*. At the one extreme, if each column is stored in a separate column group, Albis mimics a column store. On the other hand, if all columns are stored together in a single column group, it resembles a row store. In essence, the format is inspired by the DSM [23] model without mirroring columns in column groups. Horizontal partitioning is the same as sharding the table along the rows and storing them into multiple *row-groups (RGs)*. A typical row group is configurable either based on the number of rows or the size (typically a few GBs). The number and ordering of the row and column groups are inferred from their names as shown in Figure 3. Albis does not maintain any explicit indexes. Row data in various column groups are matched together implicitly by their position in the data file. The data, metadata, and file management and naming convention of Albis is similar to BigTable [22]. In the following sections, we discuss the storage format, read/write paths, support for data/schema evolution, and concerns with distributed data processing systems in detail. Table 4 shows the abridged Albis API.

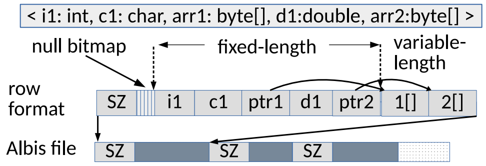


Figure 4: Albis row storage format.

3.1 Row Storage Format

After splitting the table along multiple CGs, each CG can be thought of as a table with its own schema and data. Row data from a column group is stored continuously, one after another, in a file. The Albis row format consists of four sections: a size counter, a null bitmap, a fixed-length and a variable-length section as shown in Figure 4. For a given schema, the number of fields determines the bitmap size in bytes. For example, a 23 columns schema (like TPC-DS `store_sales`) takes 3 bytes for the null bitmap. The fixed-length area is where data for fixed-length columns are stored in situ. A variable-length column data is stored in the variable-length area, and its offset and size is encoded as an 8-byte long and stored in the fixed area. With this setting, for a given schema, Albis calculates the fixed-length section size (that stays fixed, hence the name) by summing up the size of the fixed-type fields and 8×number of variable fields. For example, a schema of `<int, char, byte[], double, byte[]>` (as shown in the figure) takes one byte for bitmap, and 29 (= 4 + 1 + 8 + 8 + 8) bytes for the fixed segment. The row encoding is then prefixed by the total size of the row, including its variable segment. For a fixed-length schema (contains only fixed-length fields), Albis optimizes the row format by eschewing the size prefix as all rows are of the fixed, same size.

3.2 Writing Row Data

A writer application defines a schema and the column grouping configuration by allocating `AlbisSchema` and `AlbisColumn` objects. In the default case, all columns are stored together in a row-major format. The writer application then allocates an `AlbisWriter` object from the schema object. The writer object is responsible for buffering and formatting row data according to the storage format as described previously. Internally, the writer object manages parallel write streams to multiple CG locations, while counting the size. Once a RG size is reached, the current writers are closed, and a new set of writers in a new RG directory are allocated. Data is written and read in the multiple of *segments*. A segment is a unit of I/O buffering and metadata generation (default: 1 MB). The segment metadata includes the minimum and maximum values (if applicable), distribution

Class	Functions	Action

AlbisFileFormat	reGroup(Schema, Path, Schema, Path)
	reBalance(Path)
		re-groups the schema in the given path location
		re-balances the data in the given path location

AlbisColumn	make(String, Type, ...)	makes a column with a name and type
AlbisColGroup	make(AlbisColumn[])	makes a column group from a given list of columns

AlbisSchema	getReader(Path, Filter[], AlbisColumn[])
	getWriter(Path)
	makeSchema(ColumnGroup[])
		gets a reader for a given location, projection, and filter
		gets a writer to a given location
		builds a schema with a given list of CGs

AlbisWriter	setValue(Int, Value)
	nextRow()
		sets a value (can be null) for a given column index
		marks the current row done, and moves the pointer

AlbisReader	hasMore() and next()
	getValue(Int)
	getBinaryRow()
		implements the Scala Iterator abstraction for rows
		gets the value (can be null) for a given column index
		returns a byte[] with the encoded row

Table 4: Abridged Albis API. Apart from row-by-row, Albis also supports a vector reading interface.

of data (e.g., sorted or not), number of rows, padding information, and offset in the data file, etc. The segment metadata is used for implementing filters.

3.3 Reading Row Data

A reader application first reads the schema from the top-level directory and scans the directory paths to identify row and column groups. The reader then allocates an AlbisReader object from the schema, which internally reads in parallel from all column groups to construct the complete row data. AlbisReader implements the Iterator abstraction where the reader application can check if there are more rows in the reader and extract values. The reader object reads and processes a segment's worth of data from all column groups in parallel, and keeps the row index between them in sync. An AlbisReader object also supports a binary API where row-encoded data can be returned as a byte[] to the application.

Projection: AlbisReader takes a list of AlbisColumns that defines a projection. Internally, projection is implemented simply as re-ordering of the column indexes where certain column indexes are skipped. Naturally, the performance of the projection depends upon the column grouping. In the row-major configuration, Albis cannot avoid reading unwanted data. However, if the projected columns are grouped together, Albis only reads data files from the selected column group, thus skipping unwanted data. As we will show in Section 4.1, the implementation of projection is highly competitive with other formats.

Filtering: Albis implements two levels of filtering. The first-level of filtering happens at the segment granularity. If a filter condition is not met by the metadata, the segment reading is skipped immediately. For example, if a segment metadata contains the max value of 100 for an integer column, and a filter asks for values greater than 500, the segment is skipped. However, if the condition is not specific enough, then the rows are checked one-by-one, and only valid rows satisfying all filter conditions are returned. Currently, Albis supports null checks and ordinal filters (less than, greater than, equal to) with combinations of logical (NOT, AND and OR) operators.

3.4 Data and Schema Evolution in Albis

As described so far, the name and location of a data file plays an important role to support data and schema evolution in Albis. We now describe this in detail:

Adding Rows: Adding another row is trivial. A writer adds another row group in the directory and writes its data out. Appending to an existing row group is also possible on append-only file systems like HadoopFS. However, while adding another row, the writer cannot alter the column grouping configuration.

Deleting Rows: Deleting rows in-place is not supported as the underlying file system (HDFS) is append-only.

Adding Columns: Adding new columns is one of the most frequent operations in analytic. Adding a column at the end of the schema involves creating a column group directory (with associated data and metadata files). The ordering of row data in the newly added column is matched with the existing data, and missing row entries are marked null. Using this strategy, more than one column (as a CG) can be added at a time as well. The old schema file is read, and written again (after deleting the old one) with the updated schema.

Deleting Columns: A column delete operation in Albis falls in one of the two categories. The column(s) to be deleted is (are) either (i) stored as a separate CG; or (ii) stored with other columns. In the first case, the deletion operation is simple. The CG directory is deleted, and the schema is updated as mentioned previously. In the latter case, there are two ways Albis deletes columns. A light-weight deletion operation "marks" the column as deleted and updates the schema. The column is only marked as deleted, but the column is not removed from the schema because the column data is still stored in the storage. In order to skip the marked column data, an `AlbisReader` needs to know the type(s) of the deleted column(s). In contrast, a heavy-weight delete operation emulates a read-modify-write cycle, where the CG is read, modified, and written out again.

Maintenance Operations: Apart from the aforementioned operations, Albis supports two maintenance operations: *re-grouping* and *re-balancing*. Re-grouping is for re-configuring the column grouping. For example, due to the evolution in the workload it might be necessary to split (or merge) a CG into multiple CGs for a faster filter or projection processing. Re-balancing refers to re-distributing the data between RGs. A RG is the unit of processing for Albis tables in a distributed setting. Adding and removing column and row data can lead to a situation where data between row-groups is not balanced. This data imbalance will lead to imbalanced compute jobs. The Re-balancing operation reads the current data and re-balances the data distribution in the row groups. While executing the re-balancing, it is possible to increase the number of row groups to increase the parallelism in the system. Re-grouping can be executed at the same time as well. These operations are slow and we expect them to be executed once-in-a-while. Even though column adding and deleting is one of the frequent operations, a complete re-balancing is only required if the added columns/rows contain highly uneven values.

3.5 Distributed Processing Concerns

How does an RDPS system process input Albis files? RDPS frameworks like Spark/SQL divide work among the workers using the size as a criteria. At a first glance, a segment seems to be a perfect fit for providing equal sized work items for workers. For a static table, segments *can* be used as the quantum of processing. However, as a new column is added, often as a result of distributed processing, it is critical in what order the rows in the new column are written because indexes are implicitly encoded with the data position in a file. For example, imagine a table with two segments in a single row-group. Now, another column is added to this table using two Spark/SQL workers, each processing one segment. As there are no ordering guarantees between tasks, and each task in the data-parallel paradigm gets its own file to write, it is possible that the first task gets the second segment, but the first new column file name. This mix-up destroys the row ordering when enumerating files based on their names. However, if the whole row-group is processed only by a single task, the newly added "single" column file is ensured to have the same ordering as the current row file (including all its segments). Thus, an Albis row-group is the unit of processing for a distributed RDPS system. A single task is responsible for processing, adding and deleting columns and rows within a single row group while maintaining the implicit ordering variant of the data. As previously discussed, if necessary, the data can be re-balanced to increase the number of row-groups, hence, parallelism in the system. The schema file updates are expected to take place on a centralized node like the Spark driver.

Which column grouping to use? The recommended column grouping setting depends upon the workload. Systems like H2O [14], etc., can change the storage format dynamically based upon the workload. We consider this work beyond the current scope. However, we expect that Albis can help in this process by providing meaningful insights about accesses and I/O patterns.

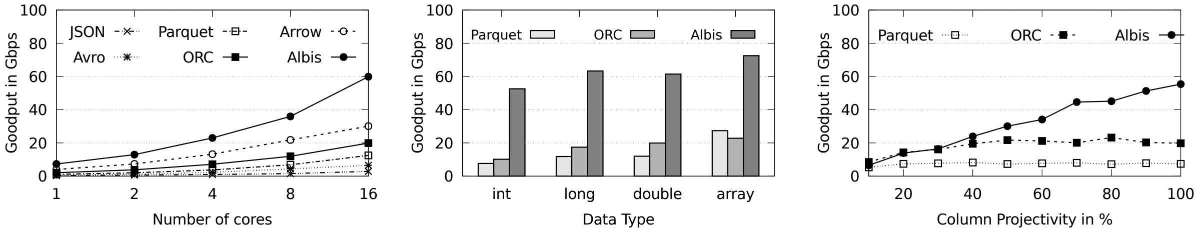
4 Evaluation

Albis is implemented in about 4k lines of Scala/Java code for Java 8. We evaluate the performance of Albis on a 4-node cluster each containing dual Xeon E5-2690 CPUs, 128GB of DDR3 DRAM, 4 enterprise-grade NVMe devices, connected via a 100 Gbps link, running Ubuntu 16.04. All numbers reported here are the average of 3 runs. We attempt to answer three fundamental questions:

- *First, does Albis deliver data access rates close to the modern networking and storage devices?* Using a set of micro-benchmarks over HDFS on 100 Gbps network and NVMe devices we demonstrate that Albis delivers read bandwidths up to 59.9 Gbps, showing a gain of $1.9 - 21.4\times$ over other file formats (Section 4.1). With Apache Crail (instead of HDFS), Albis delivers bandwidth of 85.5 Gbps from NVMe devices.
- *Secondly, does Albis accelerate the performance of data analytic workloads?* To demonstrate the effectiveness of Albis and its API, we have integrated it in Spark/SQL, and demonstrate an up to $3\times$ reduction in query runtimes. The overall TPC-DS runtime is also decreased by 25.4% (Section 4.2).
- *Lastly, what is the cost of design trade-offs of Albis, namely the cost of eschewing compression and increased look-up cost in a distributed system?* In our evaluation, Albis increases the storage cost by a margin of $1.3 - 2.7\times$ (based on the compression choice), but does not increase the load for extra block lookups. In exchange, it delivers performance improvements by a margin of $3.4 - 7.2\times$ (Section 4.3).

4.1 Micro-benchmarks

In this section, we evaluate the performance of Albis, through a series of micro-benchmarks. We focus on the



(a) (b) (c)

Figure 5: (a) Bandwidth vs. core scaling; (b) Effect of data type on performance; (c) Projectivity performance.

read because the write performance of all file formats are bottlenecked by the HDFS write bandwidth ($\sim$20 Gbps).

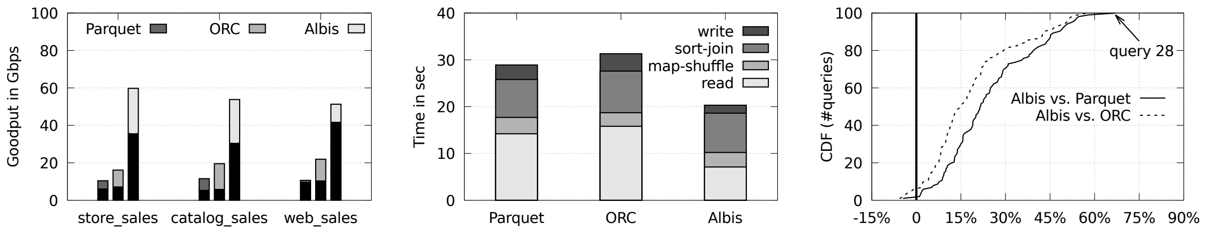
Read performance: We start by revisiting the key experiment from the beginning of the paper. Here, in Figure 5a we show the performance of Albis with respect to other file formats for reading the `store_sales` data. The x-axis shows the number of cores and the y-axis shows the data goodput performance. As shown, in comparison to other data formats, Albis delivers 1.9 (vs. Arrow) – 21.4× (vs. JSON) better read performance for reading the `store_sales` table. The performance gains of Albis can be traced down to its superior instruction utilization (due to its light-weight software stack), and cache profile. Table 5 shows the CPU profile of Albis against Parquet, ORC, and Arrow. As can be seen, Albis takes $1.2-4.1\times$ less instructions, and exhibits $1.5-3.0\times$ fewer cache misses per row. The peak data rate is at 59.9 Gbps, which is within 80% of the HDFS bandwidth. The gap between the HDFS performance and Albis is due to the parsing of schema and materialization of the data. For the sake of brevity, in the following sections we focus our effort on best performing formats, namely Parquet and ORC, for the performance evaluation. Arrow does not have native filter and projection capabilities.

Effect of schema and data types: We now focus our effort to quantify what effect a data type has on the read performance. We choose integers, longs, doubles, and arrays of byte types. For this benchmark, we store 10 billion items of each type. For the array, we randomly generate an array in the range of (1–1024) bytes. Figure 5b shows our results. The key result from this experiment is that Albis's performance is very close to the expected results. The expected result is calculated as what fraction of incoming data is TPC-DS data. As discussed in Section 3.1, each row contains an overhead of the bitmap. For a single column schema that we are testing, it takes 1 byte for the bitmap. Hence, for integers we expect $4/5^{th}$ of the HDFS read performance. In our experiments, the integer performance is 52.6 Gbps which is within 87.8% of the expected performance (59.9 Gbps). Other types also follow the same pattern. The double values are slower to materialize than longs. We are not sure about the cause of this behavior. The byte array schema delivers bandwidth very close to the HDFS read bandwidth as the bitmap overhead is amortized. With arrays, Albis delivers 72.5 Gbps bandwidth. We have only shown the performance of primitive types as higher-level types such as `timestamps, decimal`, or `date` are often encoded into the lower-level primitive types, and their materialization is highly implementation-dependent.

	Parquet	ORC	Arrow	Albis
Instructions/row	6.6K	4.9K	1.9K	1.6K
Cache-misses/row	9.2	4.6	5.1	3.0
Nanosecs/row	105.3	63.9	31.2	20.8

Table 5: Micro-architectual analysis for Parquet, ORC, Arrow, and Albis on a 16-core Xeon machine.

Projection performance: A columnar-format is a natural fit when performing a projection operation. A similar operation in a row-oriented format requires a level of redirection to materialize the values while leaving the unwanted data behind. However, a distinction must be made between a true column-oriented and PAX-alike format (e.g., Parquet). The PAX-encoding does not change the I/O pattern and amount of data read from a file system. It only helps to reduce the amount of work required (i.e., mini-joins) to materialize the projected columns. Albis's efficiency in projection depends upon the column grouping configuration. With a single column group, Albis is essentially a row store. Hence, the complete row data is read in, but only desired columns are materialized. To evaluate the projection performance, we generated a dataset with 100 integer columns and 100 million rows (total size: 40 GB). This dataset is then read in with a variable projectivity, choosing k out of 100 columns for k% projectivity. Figure 5c shows the projectivity (x-axis) and the goodput (y-axis). As shown, Albis (as a row store) always outperforms Parquet and ORC after 30% projectivity. It is only slower than ORC for 10% and 20% projectivity by a margin of 23.5% and 2.7%, respectively. We exercise caution with results as the superiority of row versus column format is a contentious topic, and gains of one over the other come from a variety of



(a)

(b)

(c)

Figure 6: Albis performance results: (a) read bandwidths (with Spark iterator in dark) of file formats for TPC-DS tables; (b) performance of the EquiJoin on two 32GB data sets; (c) CDF for performance gains for TPC-DS queries.

features that go beyond just micro-benchmarks [12].

Selectivity performance: In a real-world workload, RDPS systems often have predicates on column values. As all file formats maintain metadata about columns, they help the RDPS systems to pre-filter the input rows. However, it must be noted that often, filters are "*hints*" to file formats. A format is not expected strictly to return rows that satisfy the filters. Parquet, ORC, and Albis all maintain segment-level metadata that can be used to completely eliminate reading the segment. The performance saving depends upon the segment size. ORC also maintains metadata per 10k rows that allows it to do another level of filtering. In contrast, Albis supports strict filtering of values and hence, it avoids pushing unwanted rows into the processing pipeline. We evaluate selectivity performance on the same 100 integer column table used in the projection. The first column of the table contains integers between 0 and 100. We execute a simple SQL query "*select(*) from table where col0 <= k*", where k varies from 1 to 100, to select k% of input rows. Our results demonstrate similar gains (not shown) as the projection performance. Albis outperforms Parquet and ORC by a margin of $1.6-2.4\times$.

4.2 Workload-level Performance

For workload-level experiments, we have integrated Albis support into Spark/SQL [15] (v2.2) and evaluate its performance for an EquiJoin and the full TPC-DS query set. Naturally, input and output is only one aspect of a workload, and gains solely from a fast format are bounded by the CPU-I/O balance of the workload.

Spark/SQL data ingestion overheads: Integration into Spark/SQL entails converting incoming row data into Spark-specific format (the Iterator[InternalRow] abstraction). We start by quantifying what fraction of performance is lost due to framework-related overheads [54], which, of course, varies with the choice of the SQL framework. In Figure 6a, we show the read bandwidths for the three biggest tables in the TPC-DS dataset for ORC, Parquet, and Albis[3]. In each bar, we also show a dark bar that represents the performance observed at the Spark/SQL level. In general, from one third up to half of the performance can be lost due to framework-related overheads. The table web_sales performs the best with 89.2% of Parquet bandwidth delivered. However, it can be observed the other way around as well because the Parquet bandwidth is so low, hence, further overheads from the framework do not deteriorate it further.

Effect of the Binary API: While measuring the Spark ingestion rate, we also measure the number of live objects that the Java heap manages per second while running the experiment. For Albis we use two modes to materialize Spark's UnsafeRow either using the binary API or not. In our evaluation we find (not shown) that even without using the binary API, Albis (260.4K objs/sec) is better than Parquet (490.5K objs/sec) and ORC (266.4K objs/sec). The use of binary API futher improves Albis's performance by 4.3% to 249.2K objs/sec.

EquiJoin performance: Our first SQL benchmark is an EquiJoin operation, which is implemented as a Sort-Merge join in Spark/SQL. For this experiment, we generate two tables with a <int,byte[]> schema and 32 million rows each. The array size varies between 1 and 2kB. The total data set is around 64GB in two tables. The join operation joins on the int column, and then generates the checksum column for merged byte[] columns, which is written out. Figure 6b shows our results. The figure shows the runtime splits (y-axis) for the 4 stages of the join operation (reading in, mapping to partitions, sorting and joining partitions, and then the final write out) for Parquet, ORC, and Albis. As shown, Albis helps to reduce the read (7.1 sec for Albis) and write (1.7 sec for Albis) stages by more than $2\times$. Naturally, a file format does not improve the performance of the map and join stages, which remain constant for all three file formats. Overall, Albis improves the query run-time by 35.1% and 29.8% over ORC and Parquet, respectively.

TPC-DS performance: Lastly, we run the TPC-DS query set on the three file formats with the scaling factor

[3] Spark/SQL does not support the Arrow format yet (v2.2).

	None	Snappy	Gzip	zlib
Parquet	58.6 GB 12.5 Gbps	44.3 GB 9.4 Gbps	33.8 GB 8.3 Gbps	N/A -
ORC	72.0 GB 19.1 Gbps	47.6 GB 17.8 Gbps	N/A -	36.8 GB 13.0 Gbps
Albis	94.5 GB 59.9 Gbps	N/A -	N/A -	N/A -

Table 6: TPC-DS dataset sizes and performance.

of 100. We choose the factor 100 as it is the largest factor that we can run while keeping the shuffle data in the memory to avoid Spark's shuffle-related overheads. Figure 6c shows our results. On the y-axis, the figure shows the fraction of queries as CDF and on the x-axis it shows percentage performance gains for Albis in comparison to Parquet and ORC formats. There are two main observations here. First, for more than half of the queries, the performance gains are less than 13.8% (ORC) and 21.3% (Parquet). For 6 queries on ORC (only 1 on Parquet), the gains are even negative, however, the loss is small (-5.6%). Second, the last six queries on both file formats see more than a 50% improvement in run-times. The run-time of the query 28, which is the query with most gains, is improved by a margin of 2.3× and 3.0× for ORC and Parquet, respectively. Gains are not uniformly distributed among all queries because they depend upon what fraction of the query time is I/O bounded and what is CPU bounded (including framework related overheads). The run-times of the full TPC-DS suit with all queries is $1,850.1$ sec (Parquet), $1,723.4$ sec (ORC) and $1,379.2$ sec (Albis), representing a gain of 25.4% (over Parquet) and 19.9% (over ORC).

4.3 Efficiency of Albis

The cost of compression: With its sole focus on performance, Albis does not use any compression or encoding to reduce the data set size. This design choice means that Albis file sizes are larger than other file formats. In Table 6, we show the TPC-DS dataset (scale=100) sizes with compression options available on Parquet and ORC. As calculated from the table, due to highly efficient type-specific encoding (e.g. RLE for integers), even uncompressed Parquet and ORC datasets are $23.8 - 37.9\%$ smaller than Albis's. With compression, the gap widens to 64.2%. With the current market prices, the increased space costs 0.5$/GB on NVMe devices. However, the compressed dataset sizes also lead to significant performance loss when reading the dataset (also shown in the table). As we have shown in Section 2, the reading benchmarks are CPU bounded even without compression. Hence, adding additional routines to decompress incoming data only steals cycles from an already saturated CPU. In comparison to compressed bandwidths, Albis delivers $3.4 - 7.2\times$ higher bandwidths. One potential avenue to recover the lost storage efficiency is to leverage advanced hardware-accelerated features like deduplication and block compression further down the storage stack. However, we have not explored this avenue in detail yet.

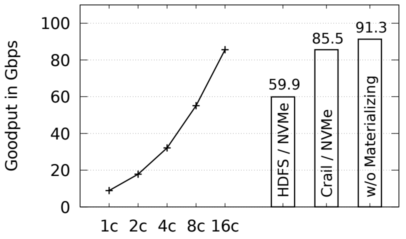


Figure 7: Albis performance on Crail with NVMeF.

Load on the distributed system: One concern with the increased number of files and the storage capacity of data sets is that they increase RPC pressure on the namenode. The number of RPCs to the namenode depends upon the number of files and blocks within a file. With an HDFS block size of 128 MB, the most efficient dataset from TPC-DS takes 271 blocks (33.8 GB with Parquet and gzip). In comparison, Albis's dataset takes 756 blocks. The lookup cost increase for hundreds of blocks is marginal for HDFS. Nonetheless, we are aware of the fact that these factors will change with the scale.

Delivering 100 Gbps bandwidth: For our final experiment, we try to answer the question what it would take to deliver 100 Gbps bandwidth for Albis. Certainly, the first bottleneck is to improve the base storage layer performance. The second factor is to improve the data density. For example, the `store_sales` table on Albis has the data density of 93.9% (out of 100 bytes read from the file system). On top of this, the effective bandwidth on 100 Gbps link is 98.8 Gbps, that gives us the upper bounds for the performance at 92.8 Gbps that we hope to achieve with the Albis `store_sales` table on a 100 Gbps link. To test the peak performance, we port Albis to Apache Crail with its NVMeF tier [8, 52]. Crail is an Apache project that integrates high-performance network (RDMA, DPDK) and storage (NVMeF, SPDK) stacks in the Apache data processing ecosystem. The peak bandwidth of Crail from NVMe devices is 97.8 Gbps [46]. Figure 7 shows our results. In the left half of the figure it shows the scaling performance of Albis on Crail from 1 core performance (8.9 Gbps) to 16 cores (85.5 Gbps). In comparison, the right half of the figure shows the performance of HDFS/NVMe at 59.9 Gbps and Crail/NVMe at 85.5 Gbps. The last bar shows the performance of Albis if the benchmark does not materialize Java object values. In this configuration, Albis on

Crail delivers 91.3 Gbps, which is within 98.4% of the peak expected performance of 92.7 Gbps.

5 Related Work

Storage Formats in Databases: N-ary Storage Model (NSM) stores data records contiguously in a disk page, and uses a record offset table at the end of the page [47]. Decomposition Storage Model (DSM) [23] proposes to split an n-column relation into n sub-relations that can be stored independently on a disk. NSM is considered good for transactions, whereas, DSM is considered suitable for selection and projection-heavy analytical workloads. A series of papers have discussed the effect of NSM and DSM formats on the CPU efficiency and query execution [27, 55, 12]. Naturally, there is no one size that fits all. Ailamaki et al. [13] propose the Partition Attributes Across (PAX) format that combines the benefits of these two for a superior cache performance. The Fractured Mirrors design proposes maintaining both NSM and DSM formats on different copies of data [48]. Jindal et al. propose the Trojan data layout that does workload-driven optimizations for data layouts within replicas [32]. The seminal work from Abadi et al. demonstrates that a column-storage must be augmented by a right query processing strategy with late materialization, batch processing, etc., for performance gains [12]. Various state of the art database systems have been proposed that take advantage of these strategies [51, 19]. In comparison so far, the focus of Albis has been on a *light-weight* file format that can faithfully reflect the performance of the storage and networking hardware.

File Formats in the Cloud: Many parts of the data format research from databases have found its way into commodity, data-parallel computing systems as well. Google has introduced SSTable, an on-disk binary file format to store simple immutable key-value strings [22]. It is one of the first external file formats used in a large-scale table storage system. RCFile [28] is an early attempt to build a columnar store on HDFS. RCFiles do not support schema evolution and have inefficient I/O patterns for MapReduce workloads. To overcome these limitations, Floratou et al. propose the binary columnar-storage CIF format for HDFS [25]. However, in order to maintain data locality, they require a new data placement policy in HDFS. Hadoop-specific column-storage issues like column placement and locality are discussed in detail by [30, 25]. The more popular file formats like Parquet [10] (uses Dremel's column encoding [38]) and ORC [9], etc., are based on the PAX format. Albis's column grouping and row-major storage format match closely with Yahoo's Zebra [4, 39]. However, Zebra does not support filter pushdown or statistics like Albis. Apache CarbonData is an indexed columnar data format for fast analytics on big data platforms [7]. It shares similarities with the Arrow/Parquet project. However, due to its intricate dependencies on Spark, we could not evaluate it independently. Historically, the priorities of these file formats have been I/O efficiency (by trading CPU cycles) and then performance, in that order. However, as we have demonstrated in this paper, the performance of these file formats are in dire need of revision.

High-Performance Hardware: Recently, there has been a lot of interest in integrating high-performance hardware into data processing systems [17, 18]. Often, the potential performance gains from modern hardware are overshadowed by the thick software/CPU stack that is built while holding the decades old I/O assumptions [40, 54]. This pathology manifests itself as "system being CPU-bounded", even for many I/O-bound jobs [20, 43]. A natural response to this situation is to add more resources, which leads to a significant loss in efficiency [37]. In this work, we have shown that by re-evaluating the fundamental assumptions about the nature of I/O and CPU performances, we can build efficient and fast systems - a sentiment echoed by the OS designers as well [45]. Recently, Databricks has also designed its optimized caching format after finding out about the inadequate performance of file formats on NVMe devices [36]. However, details of the format are not public.

6 Conclusion

The availability of high-performance network and storage hardware has fundamentally altered the performance balance between the CPU and I/O devices. Yet, many assumptions about the nature of I/O are still rooted in the hardware of the 1990s. In this paper, we have investigated one manifestation of this situation in the performance of external file formats. Our investigation on 100 Gbps network and NVMe devices reveals that due to the excessive CPU and software involvement in the data access path, none of the file formats delivered performance close to what is possible on modern hardware. Often, CPU cycles are traded for storage efficiency. We have presented Albis, a light-weight, high-performance file format. The key insight in the design of Albis is that by foregoing the assumptions and re-evaluating the CPU-I/O work division in the file format, it is possible to build a high-performance *balanced* system. In the process of designing Albis, we have also presented an extensive evaluation of popular file formats on modern high-performance hardware. We demonstrate that Albis delivers performance gains in the order of $1.9-21.4\times$; superior cache and instruction profile; and its integration in Spark/SQL shows TPC-DS queries acceleration up to a margin of $3\times$. Encouraged by this result, we are exploring applicability of Albis with multiple frameworks.

References

[1] Amazon EC2 I3 Instances: Storage optimized for high transaction workloads. https://aws.amazon.com/ec2/instance-types/i3/. [Online; accessed Jan-2018].

[2] AWS News Blog, Elastic Network Adapter - High Performance Network Interface for Amazon EC2. https://aws.amazon.com/blogs/aws/elastic-network-adapter-high-performance-network-interface-for-amazon-ec2/. [Online; accessed Jan-2018].

[3] Microsoft Azure: High performance compute VM sizes, RDMA-capable instances. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-hpc#rdma-capable-instances. [Online; accessed Jan-2018].

[4] Pig 0.8 Documentation, Zebra Overview. https://pig.apache.org/docs/r0.8.1/zebra_overview.html. [Online; accessed Jan-2018].

[5] Apache Arrow - Powering Columnar In-Memory Analytics. https://arrow.apache.org/, 2017. [Online; accessed Jan-2018].

[6] Apache Avro. https://avro.apache.org/, 2017. [Online; accessed Jan-2018].

[7] Apache CarbonData. http://carbondata.apache.org/index.html, 2017. [Online; accessed Jan-2018].

[8] Apache Crail (Incubating) - High-Performance Distributed Data Store. http://crail.incubator.apache.org/, 2017. [Online; accessed Jan-2018].

[9] Apache ORC - High-Performance Columnar Storage for Hadoop. https://orc.apache.org/, 2017. [Online; accessed Jan-2018].

[10] Apache Parquet. https://parquet.apache.org/, 2017. [Online; accessed Jan-2018].

[11] JSON (JavaScript Object Notation). http://www.json.org/, 2017. [Online; accessed Jan-2018].

[12] Abadi, D. J., Madden, S. R., and Hachem, N. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada, 2008), SIGMOD '08, ACM, pp. 967–980.

[13] Ailamaki, A., DeWitt, D. J., Hill, M. D., and Skounakis, M. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 2001), VLDB '01, Morgan Kaufmann Publishers Inc., pp. 169–180.

[14] Alagiannis, I., Idreos, S., and Ailamaki, A. H2o: A hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA, 2014), SIGMOD '14, ACM, pp. 1103–1114.

[15] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia, 2015), SIGMOD '15, ACM, pp. 1383–1394.

[16] Barroso, L., Marty, M., Patterson, D., and Ranganathan, P. Attack of the Killer Microseconds. *Commun. ACM 60*, 4 (Mar. 2017), 48–54.

[17] Barthels, C., Loesing, S., Alonso, G., and Kossmann, D. Rack-Scale In-Memory Join Processing Using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia, 2015), SIGMOD '15, ACM, pp. 1463–1475.

[18] Binnig, C., Crotty, A., Galakatos, A., Kraska, T., and Zamanian, E. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow. 9*, 7 (Mar. 2016), 528–539.

[19] Boncz, P. A., Manegold, S., and Kersten, M. L. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK* (1999), pp. 54–65.

[20] Canali, L. Performance Analysis of a CPU-Intensive Workload in Apache Spark. https://db-blog.web.cern.ch/blog/luca-canali/2017-09-performance-analysis-cpu-intensive-workload-apache-spark, 2017. [Online; accessed Jan-2018].

[21] Caulfield, A. M., Coburn, J., Mollov, T., De, A., Akel, A., He, J., Jagatheesan, A., Gupta, R. K., Snavely, A., and Swanson, S. Understanding the Impact of Emerging

Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11.

[22] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington, 2006), OSDI '06, USENIX Association, pp. 205–218.

[23] COPELAND, G. P., AND KHOSHAFIAN, S. N. A Decomposition Storage Model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data* (Austin, Texas, USA, 1985), SIGMOD '85, ACM, pp. 268–279.

[24] COSTEA, A., IONESCU, A., RĂDUCANU, B., SWITAKOWSKI, M., BÂRCA, C., SOMPOLSKI, J., ŁUSZCZAK, A., SZAFRAŃSKI, M., DE NIJS, G., AND BONCZ, P. VectorH: Taking SQL-on-Hadoop to the Next Level. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA, 2016), SIGMOD '16, ACM, pp. 1105–1117.

[25] FLORATOU, A., PATEL, J. M., SHEKITA, E. J., AND TATA, S. Column-oriented Storage Techniques for MapReduce. *Proc. VLDB Endow. 4*, 7 (Apr. 2011), 419–429.

[26] GUZ, Z., LI, H. H., SHAYESTEH, A., AND BALAKRISHNAN, V. NVMe-over-fabrics Performance Characterization and the Path to Low-overhead Flash Disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference* (Haifa, Israel, 2017), SYSTOR '17, ACM, pp. 16:1–16:9.

[27] HARIZOPOULOS, S., LIANG, V., ABADI, D. J., AND MADDEN, S. Performance Tradeoffs in Read-optimized Databases. In *Proceedings of the 32Nd International Conference on Very Large Data Bases* (Seoul, Korea, 2006), VLDB '06, VLDB Endowment, pp. 487–498.

[28] HE, Y., LEE, R., HUAI, Y., SHAO, Z., JAIN, N., ZHANG, X., AND XU, Z. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering* (Washington, DC, USA, 2011), ICDE '11, IEEE Computer Society, pp. 1199–1208.

[29] HUAI, Y., CHAUHAN, A., GATES, A., HAGLEITNER, G., HANSON, E. N., O'MALLEY, O., PANDEY, J., YUAN, Y., LEE, R., AND ZHANG, X. Major Technical Advancements in Apache Hive. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA, 2014), SIGMOD '14, ACM, pp. 1235–1246.

[30] HUAI, Y., MA, S., LEE, R., O'MALLEY, O., AND ZHANG, X. Understanding Insights into the Basic Structure and Essential Issues of Table Placement Methods in Clusters. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 1750–1761.

[31] IBM. IBM InfoSphere BigInsights Version 3.0, File formats supported by Big SQL. `https://www.ibm.com/support/knowledgecenter/SSPT3X_3.0.0/com.ibm.swg.im.infosphere.biginsights.dev.doc/doc/biga_fileformats.html`, 2017. [Online; accessed Jan-2018].

[32] JINDAL, A., QUIANÉ-RUIZ, J.-A., AND DITTRICH, J. Trojan data layouts: Right shoes for a running elephant. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (Cascais, Portugal, 2011), SOCC '11, ACM, pp. 21:1–21:14.

[33] KEDIA, S., WANG, S., AND CHING, A. Apache Spark @Scale: A 60 TB+ production use case. `https://code.facebook.com/posts/1671373793181703/apache-spark-scale-a-60-tb-production-use-case/`, 2016. [Online; accessed Jan-2018].

[34] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. ReFlex: Remote Flash == Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China, 2017), ASPLOS '17, ACM, pp. 345–359.

[35] KORNACKER, M., BEHM, A., BITTORF, V., BOBROVYTSKY, T., CHING, C., CHOI, A., ERICKSON, J., GRUND, M., HECHT, D., JACOBS, M., JOSHI, I., KUFF, L., KUMAR, D., LEBLANG, A., LI, N., PANDIS, I., ROBINSON, H., RORKE, D., RUS, S., RUSSELL, J., TSIROGIANNIS, D., WANDERMAN-MILNE, S., AND YODER, M. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilo-*

mar, CA, USA, January 4-7, 2015, Online Proceedings (2015).

[36] LUSZCZAK, A., SZAFRANSKI, M., SWITAKOWSKI, M., AND XIN, R. Databricks Runtimes New DBIO Cache Boosts Apache Spark Performance: why NVMe SSDs improve caching performance by 10x. `https://databricks.com/blog/2018/01/09/databricks-runtimes-new-dbio-cache-boosts-apache-spark-performance.html`, 2018. [Online; accessed Jan-2018].

[37] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, 2015), USENIX Association.

[38] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive Analysis of Web-scale Datasets. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 330–339.

[39] MUNIR, R. F., ROMERO, O., ABELLÓ, A., BILALLI, B., THIELE, M., AND LEHNER, W. Resilientstore: A heuristic-based data format selector for intermediate results. In *Model and Data Engineering - 6th International Conference, MEDI 2016, Almería, Spain, September 21-23, 2016, Proceedings* (2016), pp. 42–56.

[40] NANAVATI, M., SCHWARZKOPF, M., WIRES, J., AND WARFIELD, A. Non-volatile Storage. *Queue 13*, 9 (Nov. 2015), 20:33–20:56.

[41] ORACLE. Oracle Big Data SQL Reference. `https://docs.oracle.com/cd/E55905_01/doc.40/e55814/bigsqlref.htm`, 2017. [Online; accessed Jan-2018].

[42] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The RAMCloud Storage System. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[43] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA, 2015), NSDI'15, USENIX Association, pp. 293–307.

[44] PETER, S., LI, J., WOOS, D., ZHANG, I., PORTS, D. R. K., ANDERSON, T., KRISHNAMURTHY, A., AND ZBIKOWSKI, M. Towards High-performance Application-level Storage Management. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems* (Philadelphia, PA, 2014), HotStorage'14, USENIX Association.

[45] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO, 2014), OSDI'14, USENIX Association, pp. 1–16.

[46] PFEFFERLE, J. Apache Crail Storage Performance – Part II: NVMf. `http://crail.incubator.apache.org/blog/2017/08/crail-nvme-fabrics-v1.html`, 2017. [Online; accessed Jan-2018].

[47] RAMAKRISHNAN, R., AND GEHRKE, J. *Database Management Systems, Chapter 9.6 Page Formats*, 3^{rd} ed. McGraw-Hill, Inc., 2003.

[48] RAMAMURTHY, R., DEWITT, D. J., AND SU, Q. A Case for Fractured Mirrors. In *Proceedings of the 28th International Conference on Very Large Data Bases* (Hong Kong, China, 2002), VLDB '02, VLDB Endowment, pp. 430–441.

[49] RUDNYTSKIY, V. Loading sample data from different file formats in SAP Vora. `https://www.sap.com/developer/tutorials/vora-zeppelin-load-file-formats.html`, 2017. [Online; accessed Jan-2018].

[50] SCHUEPBACH, A., AND STUEDI, P. Apache Crail Storage Performance – Part III: Metadata. `http://crail.incubator.apache.org/blog/2017/11/crail-metadata.html`, 2017. [Online; accessed Jan-2018].

[51] STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway, 2005), VLDB '05, VLDB Endowment, pp. 553–564.

[52] STUEDI, P., TRIVEDI, A., PFEFFERLE, J., STOICA, R., METZLER, B., IOANNOU, N., AND KOLTSIDAS, I. Crail: A High-Performance I/O

Architecture for Distributed Data Processing. *IEEE Bulletin of the Technical Committee on Data Engineering 40*, 1 (March 2017), 40–52.

[53] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow. 2*, 2 (Aug. 2009), 1626–1629.

[54] TRIVEDI, A., STUEDI, P., PFEFFERLE, J., STOICA, R., METZLER, B., KOLTSIDAS, I., AND IOANNOU, N. On The [Ir]relevance of Network Performance for Data Processing. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (Denver, CO, 2016), USENIX Association.

[55] ZUKOWSKI, M., NES, N., AND BONCZ, P. DSM vs. NSM: CPU Performance Tradeoffs in Block-oriented Query Processing. In *Proceedings of the 4th International Workshop on Data Management on New Hardware* (Vancouver, Canada, 2008), DaMoN '08, ACM, pp. 47–54.

Notes: IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates Other products and service names might be trademarks of IBM or other companies.

Litz: Elastic Framework for High-Performance Distributed Machine Learning

Aurick Qiao[1,2], Abutalib Aghayev[2], Weiren Yu[1,3], Haoyang Chen[1],
Qirong Ho[1], Garth A. Gibson[2,4], Eric P. Xing[1,2],
[1]*Petuum, Inc.* [2]*Carnegie Mellon University* [3]*Beihang University* [4]*Vector Institute*

Abstract

Machine Learning (ML) is an increasingly popular application in the cloud and data-center, inspiring new algorithmic and systems techniques that leverage unique properties of ML applications to improve their distributed performance by orders of magnitude. However, applications built using these techniques tend to be static, unable to elastically adapt to the changing resource availability that is characteristic of multi-tenant environments. Existing distributed frameworks are either inelastic, or offer programming models which are incompatible with the techniques employed by high-performance ML applications.

Motivated by these trends, we present Litz, an elastic framework supporting distributed ML applications. We categorize the wide variety of techniques employed by these applications into three general themes — stateful workers, model scheduling, and relaxed consistency — which are collectively supported by Litz's programming model. Our implementation of Litz's execution system transparently enables elasticity and low-overhead execution.

We implement several popular ML applications using Litz, and show that they can scale in and out quickly to adapt to changing resource availability, as well as how a scheduler can leverage elasticity for faster job completion and more efficient resource allocation. Lastly, we show that Litz enables elasticity without compromising performance, achieving competitive performance with state-of-the-art non-elastic ML frameworks.

1 Introduction

Modern clouds and data-centers are multi-tenant environments in which the set of running jobs and available resources (CPU, memory, etc.) at any given time are constantly changing [5, 45, 27]. At the same time, Machine Learning (ML) is quickly becoming a dominant application among modern distributed computing workloads. It is therefore highly desirable for ML applications executing in such an environment to be *elastic*, being able to opportunistically use additional resources when offered, and gracefully release acquired resources when requested. Elasticity is beneficial for both the individual job and for the cluster as a whole. An elastic job can make use of idle resources to complete within a shorter amount of time, and still make progress when some of its resources are removed. A cluster-wide job scheduler can dynamically re-allocate resources to speed up urgent real-time or interactive jobs, and ensure fairness by preventing jobs from holding highly contested resources for long periods of time.

Recent advancements in algorithmic and systems techniques for distributed ML applications have improved their performance by an order of magnitude or more. New algorithms such as AdaptiveRevision [39], NOMAD [42], and LightLDA [55] can better scale in distributed environments, possessing favorable properties such as staleness tolerance [39, 28], lock-free execution [42, 56], and structure-aware parallelization [20, 55]. Systems and frameworks such as GraphLab [38], Petuum [53], Adam [15], and various parameter servers [36, 28] are able to support and exploit these properties to achieve even higher performance, using techniques such as bounded-staleness consistency models [17], structure-aware scheduling [33], bandwidth management/re-prioritization [50], and network message compression [52, 15].

Although significant work is being done to push the boundaries of distributed ML in terms of performance and scalability, there has not been as much focus on elasticity, thus limiting the resource adaptability of ML applications in real-world computing environments.

General-purpose distributed frameworks such as Hadoop [1] and Spark [57] are well integrated with cloud and data-center environments, and are extensively used for running large-scale data processing jobs. They are designed to support a wide spectrum of conventional tasks—including SQL queries, graph computations, and sorting and counting—which are typically transaction-oriented and rely on deterministic execution. However, their programming models are incompatible with the algorithmic and systems techniques employed by distributed ML applications, abstracting away necessary details such as input data partitioning, computation scheduling, and consistency of shared memory access. As a result, the performance of ML applications built using these frameworks fall short of standalone implementations by two orders of magnitude or more [51].

Consequently, distributed ML applications are often implemented without support from elastic frameworks, resulting in jobs that hold a rigid one-time allocation of cluster resources from start to finish [50, 33, 56, 15]. The lack of an elastic framework, along with a suitable programming model which can support the various distributed ML techniques, is a key roadblock for implementing elastic ML applications.

Although the algorithmic and systems techniques employed by these standalone applications are diverse, they typically arise from only a few fundamental properties of ML that can be collectively supported by an elastic ML framework. This observation exposes an opportunity to design a framework that is able to support a large variety of distributed ML techniques by satisfying a smaller set of more general requirements. We summarize these properties of ML and how they guide the design of an elastic framework below, and further elaborate on them in Sec. 2. parent First, ML computations exhibit a wide variety of memory access patterns. Some mutable state may be accessed when processing each and every entry of a dataset, while other state may only be accessed when processing a single data entry. To improve locality of access, ML applications explicitly co-locate mutable model parameters with immutable dataset entries [55]. Each worker machine in the computation may contain a non-trivial amount of mutable state, which needs to be properly managed under an elastic setting.

Second, ML models contain a wide variety of dependency structures. Some sets of model parameters may safely be updated in parallel, while other sets of parameters must be updated in sequence. Guided by these dependency structures, ML applications carefully schedule their model updates by coordinating tasks across physical worker machines [20]. An elastic ML framework should abstract the physical cluster away from applications while still providing enough flexibility to support this type of task scheduling.

Furthermore, ML algorithms are often iterative-convergent and robust against small errors. Inaccuracies occurring in their execution are automatically corrected during later stages of the algorithm. Distributed ML applications have been able to attain higher performance at no cost to correctness by giving up traditionally desirable properties such as deterministic execution and consistency of memory access [28]. Framework mechanisms for elasticity should not rely on a programming model that restricts this way of exploiting the error-tolerance of ML algorithms.

Thus, to efficiently support ML applications, an elastic ML framework should support **stateful workers**, **model scheduling**, and **relaxed consistency**. It should provide an expressive programming model allowing the application to define a custom scheduling strategy and to specify how the consistency of memory accesses can be relaxed under it. Then, it should correctly execute this strategy within the specified consistency requirements, while gracefully persisting and migrating application state regardless of its placement with respect to input data.

Motivated by the needs and opportunities for elasticity of ML applications, we designed and implemented *Litz*[1], an elastic framework for distributed ML that provides a programming model supporting stateful workers, model scheduling and relaxed consistency.

Litz enables low-overhead elasticity for high-performance ML applications. When physical machines are added to or removed from an active job, state and computation are automatically re-balanced across the new set of available machines without active participation by the application. Litz's programming model can express key distributed ML techniques such as stateful workers, model scheduling and relaxed consistency, allowing high-performance ML applications to be implemented. Furthermore, a cluster job scheduler can leverage Litz's elasticity to achieve faster job completion under priority scheduling, and optimize resource allocation by exploiting inherent resource variability of ML algorithms.

Our main contributions are:

1. **Event-driven Programming Model for ML:** Litz exposes an event-driven programming model that cleanly separates applications from the physical cluster they execute on, enabling stateful workers and allowing the framework to transparently manage application state and computation during elastic events. Computation is decomposed into *micro-tasks* which have shared access to a distributed parameter server.

2. **Task-driven Consistency Model for ML:** Micro-tasks can be scheduled according to dependencies between them, allowing the application to perform model scheduling. Access to the parameter server is controlled by a consistency model in which a micro-task always observes all updates made by its dependencies, while having intentionally weak guarantees between independent micro-tasks.

3. **Optimized Elastic Execution System:** Litz's execution system transparently re-balances workload during scaling events without active participation from the application. It exploits Litz's programming and consistency models to implement optimizations that reduce system overhead, allowing applications using Litz to be as efficient as those using non-elastic execution systems.

The rest of this paper is organized as follows. In Sec. 2, we review ML algorithm properties and opportunities for elasticity, while Sec. 3 and Sec. 4 describes the Litz design and optimizations. In Sec. 5, we evaluate the effectiveness of Litz's optimizations in the distributed elastic setting, as well as its performance versus two other ML frameworks that are specialized to certain ML optimization techniques. Sec. 6 reviews related work, and Sec. 7 concludes the paper with a discussion towards future work.

2 Background

While ML algorithms come in many forms (e.g. matrix factorization, topic models, factorization machines, deep neural networks), nearly all of them share the following commonalities: (1) they possess a loss or objective function $\mathscr{L}(A,\mathscr{D})$, defined over a vector (or matrix) of model parameters A and collection of input data $\mathscr{D}$, and which measures how well the model parameters A fit the data $\mathscr{D}$; (2) their goal is to find a value of A that maximizes (or alternatively, minimizes) the objective $\mathscr{L}(A,\mathscr{D})$, via an *iterative-convergent* procedure that repeatedly executes a set of update equations, which

[1] Meant to evoke the strings of a harp, sounding out as many or as few. Litz is short for "Wurlitzer", a well-known harp maker.

gradually move A towards an optimal value (i.e. hill-climbing). These update equations follow the generic form

$$A^{(t)} = A^{(t-1)} + \Delta(A^{(t-1)}, \mathscr{D}), \qquad (1)$$

where $A^{(t)}$ is the vector (or matrix) of model parameters at iteration t, and $\Delta()$ is a function that computes updates to A using the previous value $A^{(t-1)}$ and the input data $\mathscr{D}$. The remainder of this section provides detailed background on specific properties of ML programs, and then presents two popular ML applications (Multinomial Logistic Regression and Latent Dirichlet Allocation) which we shall use as examples throughout this paper and as the subjects of our evaluation.

2.1 Data-parallelism and Parameter Server

Arising from the iid (independent and identically distributed) assumption on input data, the update function Δ can often be decomposed as

$$\Delta(A, \mathscr{D}) = \sum_{i=1}^{P} \Delta_i(A, \mathscr{D}_i), \qquad (2)$$

where $\mathscr{D}_1, \ldots, \mathscr{D}_P$ partition the input data $\mathscr{D}$ and each Δ_i computes a partial update using $\mathscr{D}_i$ which, when aggregated, form the final update Δ. This allows each update to be calculated in a data-parallel fashion with input data and update calculations distributed across a cluster of workers.

Parameter Server: Eq. 2 shows that the model parameters A are used by the calculations of every partial update Δ_i. In a data-parallel setting it is natural to place the model parameters in a shared location accessible by every machine, known as a *parameter server*. Typically, implementations of this architecture consists of two types of nodes: 1) worker nodes which partition the input data and calculate partial updates and 2) parameter server nodes which partition the model parameters and aggregate/apply the partial updates sent by worker nodes. The parameter server architecture has proven to be a near-essential component of efficient distributed ML and is used in numerous applications and frameworks [50, 18, 36, 28].

Stateful Workers: Even though the model term A appears in the calculations of each partial update, not all of it is necessarily used. In particular, there may be parts of the model which are only used when processing a single partition $\mathscr{D}_i$ of the input data. A large class of examples includes non-parametric models, whose model structures are not fixed but instead depends on the input data itself, typically resulting in model parameters being associated with each entry in the input data. In such applications, it is preferable to co-locate parts of the model on worker nodes with a particular partition of input data so they can be accessed and updated locally rather than across a network. This optimization is especially essential when the input data is large and accesses to such associated model parameters far outnumber accesses to shared model parameters. It also means that workers are *stateful*, and an elastic ML system that supports this optimization needs to preserve worker state during elastic resource re-allocation.

2.2 Error Tolerance & Relaxed Consistency

ML algorithms have several well-established and unique properties, including *error-tolerance*: even if a perturbation or noise ε is added to the model parameters in every iteration, i.e. $A^{(t)} = A^{(t-1)} + \Delta(A^{(t-1)}, \mathscr{D}) + \varepsilon$, the ML algorithm will still converge correctly provided that ε is limited or bounded.

Bounded Staleness Consistency: An important application of error tolerance is bounded staleness consistency models [28, 17, 13], which allow stale model parameters to be used in update computations, i.e. $A^{(t)} = A^{(t-1)} + \Delta(A^{(t-s)}, \mathscr{D})$, where $1 \le s \le k$ for small values of k. ML algorithms that use such consistency models are able to (1) execute in a partially asynchronous manner without sacrificing correctness, thus mitigating the effect of stragglers or slow workers [16, 25]; and (2) reduce the effect of network bottlenecks caused by synchronization by allowing cached parameter values to be used. Stale-Synchronous Parallel (SSP) [28] is such a consistency model, under which a set of distributed workers may read cached values from a shared parameter server as long as their staleness do not exceed a fixed limit.

Staleness-aware ML Algorithms: Beyond simply applying bounded staleness consistency to existing algorithms, the ML community has developed new staleness-aware algorithms [39, 58, 55, 12, 29, 10, 37] which modify each update $\Delta()$ according to the staleness s that it experiences. The modifications usually take the form of a scaling factor $\Delta() \leftarrow c\Delta()$, which are computationally light-weight and do not create new bottlenecks. In the presence of staleness, these algorithms converge up to an order of magnitude faster than their non-staleness-aware counterparts.

2.3 Dependencies and Model Scheduling

Another key property of ML algorithms is the presence of implicit *dependency structures*: supposing A_1 and A_2 are different elements of A, then updating A_1 before A_2 does not necessarily yield the same result as updating A_2 before A_1; whether this happens or not depends on the algebraic form of $\mathscr{L}()$ and $\Delta()$. As a consequence, the convergence rate and thus the running time of ML algorithms can be greatly improved through careful scheduling of parallel model parameter updates.

Dependency-aware ML Algorithms: Like the many existing staleness-aware algorithms that exploit error tolerance, there is a rich set of algorithms that use dependency structures in their models to perform better scheduling of updates [44, 55, 20, 18, 35, 49, 38]. A typical example is to partition the model into subsets, where the parameters inside a subset must be updated sequentially, but multiple subsets can be updated in parallel. Two parameters A_1 and A_2 are placed into the same subset if the strength of their dependency exceeds a threshold $\text{dep}(A_1, A_2) > \varepsilon$. As with staleness-aware algorithms, dependency-aware algorithms converge up to an order of magnitude faster than their non-dependency-aware counterparts.

3 Litz Programming Model and API

The main goal and challenge of designing Litz's programming model is striking a balance between being expressive enough to support the wide variety of proven techniques in distributed ML, while exposing enough structure in the application that the underlying execution system can take control under elastic conditions. Guided by the insights presented in Sec. 2, we describe how Litz's programming model naturally arises from the properties of ML applications, and how it enables an efficient and elastic run-time implementation. For reference, a detailed summary of Litz's API can be found in Table 1.

Input Data Over-Partitioning Across Executors: Eq. 2 shows that the input data and update calculations of ML applications can be partitioned and distributed across a number of workers, but it does not specify any particular partitioning scheme, nor does it require the number of partitions to be equal to the number of physical machines. Instead of directly assigning input data, Litz first distributes it across a set of logical *executors*, which are in turn mapped to physical machines. Elasticity is enabled by allocating more executors than physical machines and migrating excess executors to other machines as they become available. This separation also lets Litz support stateful workers by allowing executor state to be defined and mutated by the application while being treated as a black box by the run-time system.

Micro-Tasks and Parameter Server: Update calculations are decomposed into short-lived (typically shorter than 1 second) units of computation called *micro-tasks*, each of which calculates a partial update using the input data on a single executor. At the end of each micro-task, control is yielded back to the run-time system, exposing frequent opportunities for executors to be migrated. During its execution, a micro-task is granted read/update access to a global parameter server via a key-value interface (`PSGet/PSUpdate` in Table 1) and applies partial updates to model parameters by modifying application state in the executor and/or updating globally-shared values in the parameter server.

Model Scheduling and Relaxed Consistency: Litz enables both model scheduling and relaxed consistency using application-defined dependencies between micro-tasks. If micro-task A is a dependency of micro-task B, then (1) B is executed before A and (2) B observes all updates made by A. This strict ordering and consistency guarantee lets the application perform model scheduling by defining an ordering for when certain updates are calculated and applied. On the other hand, if neither A nor B is a dependency of the other, then they may be executed in any order or in parallel, and may observe none, some, or all of the updates made by the other. This critical piece of non-determinism lets the application exploit relaxed consistency models by allowing the run-time system to cache and use stale values from the parameter server between independent micro-tasks.

Micro-Task Dispatch and Completion: A common way to specify dependencies between tasks is through a directed "dependency" graph in which each vertex corresponds to a micro-task, and an arc from vertex A to vertex B means task A is a dependency of task B. However, due to a potentially large number of micro-tasks, explicitly specifying such a graph up-front may incur significant overhead. Instead, each Litz application defines a *driver* which dynamically dispatches micro-tasks during run-time via the `DispatchTask` method. When a micro-task completes, Litz invokes the `HandleTaskCompletion` method on the driver, which can then dispatch any additional micro-tasks.

Without an explicit dependency graph, Litz needs an alternative way to decide when a micro-task should be able to observe another micro-task's updates. Otherwise, its execution system does not have enough information to know when it is safe for a micro-task to use cached parameter values, thus giving up a significant opportunity for performance optimization. To overcome this issue, Litz uses the sequence of micro-task dispatch and completion events to infer causal relationships between micro-tasks, which can then be used to generate the dependencies needed to implement its cache coherence protocol. According to the following two cases:

1. If micro-task B is dispatched *before* being informed of the completion of micro-task A, then Litz infers that the completion of A *did not cause* the dispatch of B. A *is not* a dependency of B, and B may observe some, all, or none of the updates made by A.
2. If micro-task B is dispatched *after* being informed of the completion of micro-task A, then Litz infers that A *may have caused* the dispatch of B. A *may be* a dependency of B, and B will observe all updates made by A.

This consistency model is similar to Causal Memory [11], in which causally related read/write operations are observed in the same order by all nodes. We discuss how Litz's consistency model and its cache coherence protocol can be implemented efficiently in Sec. 4.

4 Litz Implementation and Optimizations

Litz is implemented in approximately 6500 lines of C++ code using the ZeroMQ [8] library for low latency communication and Boost's Context [2] library for low overhead context-switching between micro-tasks. The run-time system is comprised of a single *master thread* along with a collection of *worker threads* and *server threads*, as shown in Fig. 1. The application's driver exists in the master thread and its executors exist in the worker threads. The key/value pairs comprising the parameter server are distributed across a set of logical *PSshards* stored in the server threads. Additional worker and server threads may join at any time during the computation, and the run-time system can re-distribute its load to make use of them. They may also gracefully leave the computation after signaling to the master thread and allowing their load to be transferred to other threads.

The master thread coordinates the execution of the application. First, it obtains micro-tasks from the driver

Method Name	Part Of	Defined By	Description
`DispatchInitialTasks()`	Driver	Application	Invoked by the framework upon start-up to dispatch the first set of micro-tasks.
`HandleTaskCompletion(result)`	Driver	Application	Invoked by the framework when a micro-task completes so that the driver can dispatch a new set of micro-tasks.
`DispatchTask(executor,args)`	Driver	Framework	Invoked by the application to dispatch a micro-task to the specified executor.
`RunTask(args)`	Executor	Application	Invoked by the framework to perform a micro-task on the executor.
`SignalTaskCompletion(result)`	Executor	Framework	Invoked by the application to indicate the completion of a micro-task.
`PSGet(key)`	Executor	Framework	Returns a specified value in the parameter server.
`PSUpdate(key,update)`	Executor	Framework	Applies an incremental update to a specified value in the parameter server.

Table 1: The programming interface for Litz, an application should define `DispatchInitialTasks` and `HandleTaskCompletion` on the driver, as well as `RunTask` on the executor.

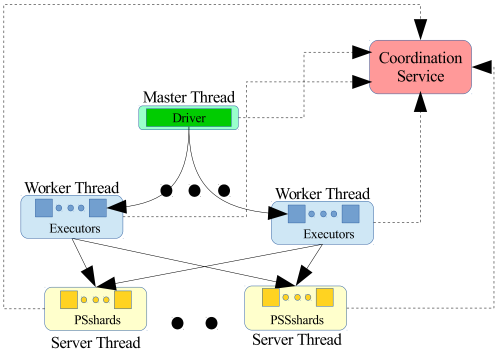


Figure 1: High-level architecture of Litz. The driver in the master thread dispatches micro-tasks to be performed by executors on the worker threads. Executors can read and update the global model parameters distributed across PSshards on the server threads.

by initially invoking `DispatchInitialTasks` and then continuously invoking `HandleTaskCompletion`, sending them to worker threads to be executed. Second, the master thread maintains the dynamic mappings between executors and worker threads, as well as between PSshards and server threads. When worker or server threads join or leave the computation, it initiates load re-distribution by sending commands to move executors between worker threads or PSshards between server threads. Third, the master thread periodically triggers a consistent checkpoint to be taken of the entire application state, and automatically restores it when a failure is detected. Each thread registers with an external coordination service such as ZooKeeper [31] or etcd [4] in order to determine cluster membership and detect failures. In order to transfer and checkpoint the driver and executors, Litz requires the application to provide serialization and de-serialization code. The programming burden on the developer is low since (1) it does not actively participate in elasticity and checkpointing, but simply invoked by the execution system when needed, and (2) third-party libraries can be used to reduce programming overhead [3].

Worker Thread Elasticity: Each worker thread maintains the state of and runs the micro-tasks for a subset of all executors. After any worker threads join the active computation, executors are moved to them from the existing worker threads (scaling out). Similarly, before any worker threads leave the active computation, executors are moved from them to the remaining worker threads (scaling in).When an executor needs to be moved, the worker thread first finishes any of its ongoing micro-tasks for that executor, buffering any other pending micro-tasks for that executor. The worker thread then sends the executor's state and its queue of buffered micro-tasks over the network to the receiving worker thread.

The transfer of the executor's input data is treated differently in the scale-in and scale-out cases. When scaling in, Litz aims to free the requested resources as quickly as possible. The input data is discarded on the originating worker thread to avoid incurring extra network transfer time, and re-loaded on the target worker thread from shared storage. When scaling out, Litz aims to make use of the new worker thread as quickly as possible. The input data is sent directly from the memory of the originating worker thread to avoid incurring extra disk read time on the target worker thread.

Parameter Server Elasticity: Similar to worker threads and executors, each server thread stores and handles the requests and updates for a subset of all PSshards, which are re-distributed before scaling in and after scaling out. However, since requests and updates are continuously being sent to each PSshard and can originate from any executor, their transfer requires a special care. In particular, a worker thread may send requests or updates to a server thread that no longer contains the target PSshard, which can occur if the PSshard has been moved but the worker thread has not yet been notified.

A naïve approach is to stop all micro-tasks on every executor, then perform the transfer, then notify all worker threads of the change, and finally resume execution. This method guarantees that requests and updates are always sent to server threads that contain the target PSshard, but incurs high overhead due to suspending the entire application. Instead, the server threads perform *request and update forwarding*, and executors are never blocked from sending a parameter request or update. When a server thread receives a message for a PSshard it no longer contains, it forwards the message to the server thread it last transferred the PSshard to. Forwarding can occur multiple times until the target PSshard is found, the request/update is performed, and the response is sent back to the originating worker thread. This way, execution of micro-tasks can proceed uninterrupted during parameter server scaling events.

Consistent Checkpoint and Recovery: To achieve fault tolerance, Litz periodically saves a checkpoint of the application to persistent storage, consisting of (1) the state of the driver, (2) the buffered micro-tasks for each executor, (3) the state of each executor, and (4) the key-value pairs stored in each PSshard. Input data is not saved, but is re-loaded from shared storage during recovery. When a failure is detected through the external coordination service, Litz triggers an automatic recovery from the latest checkpoint. The saved driver, executors, buffered micro-tasks, and parameter server values are restored, after which normal execution is resumed.

Parameter Cache Synchronization: The consistency model outlined in Sec. 3 exposes an opportunity for the runtime system to optimize execution by caching and re-using values from the parameter server instead of retrieving them over the network for each access. Specifically, a micro-task A is allowed to use a cached parameter if its value reflects all updates made by all micro-tasks that A depends on. This means that (1) multiple accesses of the same parameter by micro-task A can use the same cached value, and (2) a micro-task B whose dependencies are a subset of A's can use the same cached values that were used by A. By only using the sequence of micro-task dispatch and completion events to infer dependencies, Litz enables both (1) and (2) to be implemented efficiently. In particular, the dependencies of micro-task B are a subset of the dependencies of micro-task A if the total number of micro-tasks that have been completed when B was dispatched is at most the total number of micro-tasks that have been completed when A was dispatched.

To implement this cache coherence protocol, the master thread maintains a single monotonically increasing *version* number that is incremented each time `HandleTaskCompletion` is invoked. Whenever the driver dispatches a micro-task, the master thread tags the micro-task with the version number at that time. After micro-task A retrieves a fresh value from the parameter server, it caches the value and tags it with A's version. When micro-task B wants to access the same parameter, it first checks if its own version is less than or equal to the version of the cached value. If so, then the cached value is used; otherwise a fresh copy of the parameter is retrieved from the parameter server and tagged with B's version. A cache exists on each Litz process running at least one worker thread, so that it can be shared between different worker threads in the same process.

This cache coherence protocol allows Litz to automatically take advantage of parameter caching for applications that use bounded staleness. For example, to implement SSP (Sec. 2.2) with staleness s, all micro-tasks for iteration i are dispatched when the last micro-task for iteration $i-s-1$ is completed. Thus, every micro-task for the same iteration has the same version and share cached parameter values with each other. Since the micro-tasks for iteration i are dispatched before those for iterations between $i-s$ and $i-1$ finish (when $s \geq 1$), the values they retrieve from the parameter server may not reflect all updates made in those prior iterations, allowing staleness in the parameter values being accessed.

Parameter Update Aggregation: Updates for the same parameter value may be generated many times by different micro-tasks. Since the parameter updates in ML applications are incremental and almost always additive, they can be aggregated locally before sending to the parameter server in order to reduce network usage. To facilitate the aggregation of updates, each Litz process contains an *update log* which maps parameter keys to locally aggregated updates. Whenever a micro-task invokes `PSUpdate`, the update is first aggregated with the corresponding entry in the update log, or is inserted into the update log if the corresponding entry does not exist. Therefore, an update sent to the parameter server can be a combination of many updates generated by different micro-tasks on the same Litz process.

In order to maximize the number of updates that are locally aggregated before being sent over the network, the results of micro-tasks are not immediately returned to the master thread after they are completed. Doing this allows the updates from many more micro-tasks to be sent in aggregated form to the server threads, reducing total network usage. The update log is periodically flushed by sending all updates it contains to the server threads to be applied. After each flush, all buffered micro-task results are returned to the master thread, which then informs the driver of their completion. The period of flushing can be carefully tuned, but we find that the simple strategy of flushing only when all micro-tasks on a worker thread are finished works well in practice.

Co-operative Multitasking: Litz employs co-operative multitasking implemented using co-routines [2]. When one task is blocked on an invocation of `PSGet` waiting for a value to be returned from a server thread, the worker thread will switch to executing another micro-task that is not blocked so that useful work is still performed. Each micro-task is executed within a co-routine so that switching between them can be done with low-latency, entirely in user-space. Using co-routines provides the benefit of overlapping communication with computation, while retaining a simple-to-use, synchronous interface for accessing the parameter server from micro-tasks.

5 Evaluation

We start by evaluating Litz's elasticity mechanism and demonstrate its efficacy along several directions. First, with its parameter caching, update aggregation, and co-operative multi-tasking, Litz is able to sustain increasing numbers of executors and micro-tasks with minimal performance impact. Second, a running Litz application is able to efficiently make use of additional nodes allocated to it, accelerating its time to completion. Third, a running Litz application is able to release its nodes on request, quickly freeing them to be allocated to another job.

Next, we discuss how Litz's elasticity can be leveraged by a cluster job scheduler to (1) reduce the completion time of an ML job that yields resources to a higher-priority job, and (2) improve resource allocation by exploiting the inherent decreasing memory usage of many ML algorithms.

Lastly, we evaluate Litz's performance when executing diverse applications which make use of stateful workers, model scheduling, and relaxed consistency. With the multinomial logistic regression (MLR) application, we show that our implementation on Litz is faster than the built-in implementation in Bösen [50], a non-elastic ML system for data-parallel SSP workloads. With the latent Dirichlet allocation (LDA) application, we show that our implementation on Litz is competitive with the built-in implementation in Strads [33], a non-elastic ML system for model scheduling. Furthermore, to evaluate Litz for the special case of deep learning, we implement a deep feed-forward neural network and compare its performance with Tensorflow [9].

ML Applications: MLR and LDA are popular ML applications used for multi-class classification and topic modeling, respectively. The goal of our evaluation is to show that Litz enables elasticity for these applications at little cost to performance when compared with state-of-the-art non-elastic systems. Thus, we closely follow their implementations in Bösen and Strads, using SGD and the SSP relaxed consistency model for MLR, and block-scheduled Gibbs sampling with stateful workers for LDA. For details of these implementations of MLR and LDA, we refer readers to their descriptions in Wei *et al.* [50] and Kim *et al.* [33], respectively.

Cluster Setup: Unless otherwise mentioned, the experiments described in this section are conducted on nodes with the following specifications: 16 cores with 2 hardware threads each (Intel Xeon E5-2698Bv3), 64GiB DDR4-2133 memory, 40GbE NIC (Mellanox MCX314A-BCCT), Ubuntu 16.04 Linux kernel 4.4. The nodes are connected with each other through a 40GbE switch (Cisco Nexus 3264-Q), and access data stored on an NFS cluster connected to the same switch. Each machine runs one Litz process which contains both worker threads and server threads; the master thread is co-located with one of these processes.

Input Datasets: Unless otherwise mentioned, we run MLR on the full ImageNet ILSVRC2012 dataset [43] consisting of 1.2M images labeled using 1000 different object categories. The dataset is pre-processed using the LLC feature extraction algorithm [48], producing 21K features for each image, resulting in a post-processed dataset size of 81GB. We run LDA on a subsample of the ClueWeb12 dataset [19] consisting of 50M English web pages. The dataset is pre-processed by removing stop words and words that rarely occur, resulting in a post-processed dataset with 10B tokens, 2M distinct words, and total size of 88GB.

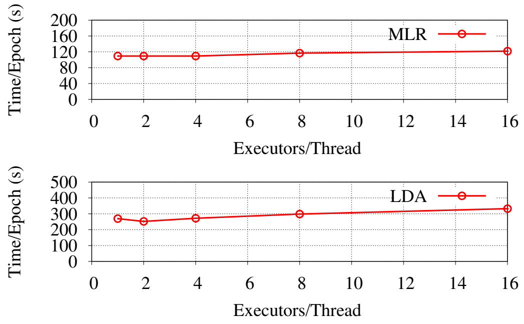


Figure 2: Average time per epoch for MLR and LDA when running with various numbers of executors per worker thread. In both cases the overhead of increasing the number of executors is insignificant. We define one epoch as performing a single pass over all input data.

5.1 Elasticity Experiments

Before discussing elastic scaling, we evaluate Litz's performance characteristics over increasing numbers of executors. The worker threads achieve elasticity by re-distributing executors amongst themselves when their numbers change, and by over-partitioning the application's state and computation across larger numbers of executors, Litz is able to scale out to larger numbers of physical cores and achieve a more balanced work assignment. Thus it is critical for Litz applications to still perform well in such configurations. We run the MLR application on 4 nodes and the LDA application on 12 nodes, varying the number of executors from 1 to 16 per worker thread. Fig. 2 shows how the throughput of each application changes when the number of executors increases. Using a single executor per worker thread as the baseline, the execution time for MLR does not noticeably change when using 4× the number of executors, and gradually increases to 1.11× the baseline when using 16× the number of executors. For LDA, the execution time initially decreases to 0.94× the baseline when using 2× the number of executors, and thereafter gradually increases to 1.23× the baseline when using 16× the number of executors. We believe the overhead introduced by increasing the number of executors is quite an acceptable trade-off for elasticity and can still be reduced with further optimizations.

5.1.1 Elastic Scale Out

As jobs finish in a multi-tenant setting and previously used resources are freed up, additional allocations can be made to a currently running job. It is therefore important for the job to be capable of effectively using the additional resources to speed up its execution. In this section, we evaluate Litz's performance characteristics when scaling a running application out to a larger number of physical nodes. We run experiments scaling MLR jobs from 4 to 8 nodes, and LDA jobs from 12 to 24 nodes. Each node runs both worker threads and server threads, so both executors and PSshards are rebalanced during scaling. The experiments for LDA in

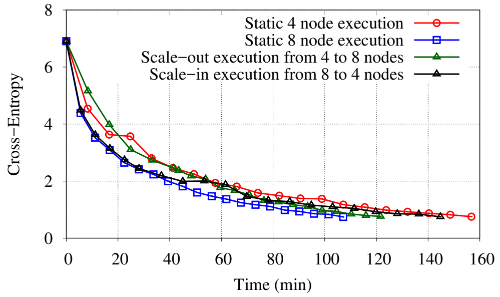


Figure 3: MLR execution on Litz with 4 nodes, with 8 nodes, with an elastic execution that scales out from 4 nodes to 8 nodes, and with an elastic execution that scales in from 8 nodes to 4 nodes. For the scale-out execution, the nodes are added at about 40 minutes into execution. For the scale-in execution, the nodes are removed at about 30 minutes into execution.

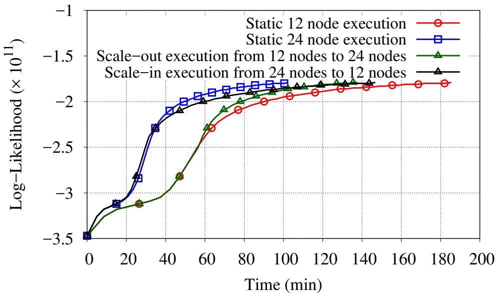


Figure 4: LDA execution on Litz with 12 nodes, with 24 nodes, and with an elastic execution that scales out from 12 nodes to 24 nodes. For the scale-out execution, the nodes are added at about 55 minutes into execution. For the scale-in execution, the nodes are removed at about 33 minutes into execution.

this section were performed using m4.4xlarge instances on AWS EC2, each with 16 vCPUs and 64GiB of memory.

To evaluate the speed-up achieved, we compare our scale-out experiments with static executions of the applications using both the pre-scaling number of nodes and the post-scaling number of nodes. Fig. 3 shows the convergence plots for MLR, 4 new nodes added after ≈40min of execution. The static 4 node execution completes in ≈157min while the scale-out execution completes in ≈122min, resulting in a 22% shorter total run-time. Fig. 4 shows the convergence plots for LDA, 12 new nodes added after ≈55min of execution. The static 12 node execution completes in ≈183min while the scale-out execution completes in ≈134min, resulting in a 27% shorter total run-time.

5.1.2 Ideal Scale Out

Next, we evaluate the amount of room for improvement still achievable over Litz's current scale-out performance. Following a similar construction as Pundir et al. [41], we define and compare with a simple *ideal* scale-out execution time which intuitively measures the total run-time of a job that instantly scales out and adapts to use the additional nodes. For example, consider a job that scales out from 4 to 8 nodes after completing 30% of its iterations, its ideal scale-out execution time is the sum of the time at which the scale-out was triggered and the time it takes a static 8 node execution to run the last 70% of its iterations.

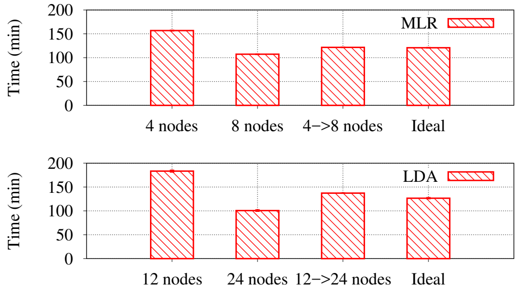


Figure 5: Static, scale-out, and ideal scale-out (See Sec. 5.1.1) execution times for MLR and LDA implemented on Litz. We scale out MLR from 4 nodes to 8 nodes, and LDA from 12 nodes to 24 nodes. Each experiment was performed several times, error bars are omitted due to their negligible size.

Fig. 5 compares the static pre-scaling, static post-scaling, scaling, and ideal execution times for both MLR and LDA. For MLR, the static 8 node execution completes in ≈107min, giving an ideal scale-out execution time of ≈121min. The actual scale-out execution time is ≈122min, indicating a less than 1% difference from the ideal. Similarly for LDA, the static 24 node execution completes in ≈101min, giving an ideal scale-out execution time of ≈127min. The actual scale-out execution time is ≈134min, indicating a 5% difference from the ideal. LDA's higher overhead stems from the large worker state that is inherent to the algorithm, which need to be serialized and sent over the network before the transferred executors can be resumed. We believe this overhead can be reduced further through careful optimization of the serialization process, by minimizing the number of times data is copied in memory and compressing the data sent over the network.

5.1.3 Elastic Scale In

As new and higher-priority jobs are submitted in a multi-tenant environment, the resource allocation for a currently running job may be reduced and given to another job. In this section, we evaluate Litz's scale-in performance based on two key factors. First, we show that Litz applications continue to make progress after scaling in, with performance comparable to the static execution on the fewer nodes. Second, we show that running Litz jobs can release resources with low latency, quickly transferring executors and PSshards away from requested nodes so that they can be used by another job. We measure the time between when the scale-in event is triggered and when the last Litz process running on a requested node exits. This represents the time an external job scheduler needs to wait before all requested resources are free to be used by another job. As with the scale-out experiments, these experiments were run using m4.4xlarge EC2 instances.

We run each experiment at least three times and report the average. Fig. 3 shows the convergence plots for MLR with the scale-in event. We start the job with 8 nodes, and remove 4 nodes ≈ 30 minutes into execution. The convergence plot closely follows the plot of 8-node static execution until the scale-in event, and the plot of 4-node static execution after that. Similarly, Fig. 4 shows the convergence plots for LDA with the scale-in event. We start the job with 24 nodes, and remove nodes ≈ 33 minutes into execution. The convergence plot closely follows the plot of 24-node static execution until the scale-in event, and the plot of 12-node static execution after that.

For MLR, the scale-in event takese 2.5 seconds on average, while for LDA the average is 43s. The low latency for MLR is due to a combination of its stateless workers and Litz's default behavior of discarding input data upon scaling in. As a result, the only state that needs to be transferred are the PSshards residing on the server threads of each requested node, which total ≈ 10MiB when split between 8 nodes. The executors in LDA, on the other hand, are stateful and contain a portion of its model parameters. When distributed across all nodes, each node contains ≈ 4.6GiB of executor state that need to be transferred away. A benchmark of cluster network showed that it can sustain a bandwidth of 2.0Gbps between pairs of machines, meaning that the 4.6GiB of LDA executor state can ideally be transfered within 20s. Nevertheless, the current transfer times are reasonable for an external scheduler to wait for. For comparison, even a pre-emptive environment like the AWS Spot Market gives users a warning time of 120s before forcefully evicting their nodes.

5.2 Elastic Scheduling

Elasticity has many potential applications in both the cloud and data-center. In the cloud, elasticity can be leveraged to take advantage of transient nodes in spot markets [26] and drastically reduce the monetary cost of renting computation resources. In the data-center, a cluster-wide scheduler can optimize resource utilization by adaptively consolidating applications into fewer physical machines [30].

We present two specific instances where the elasticity enabled by Litz can benefit job scheduling. First, when a high-priority job needs to be scheduled, an elastic ML application can avoid preemption by cooperatively releasing resources. Second, the inherent resource variability of many ML applications allow Litz to automatically release memory throughout the lifetime of an ML job, freeing resources to be used by other jobs. Serious design and implementation of such a scheduler and its policies is deserving of thorough investigation, which we leave for future work.

5.2.1 Priority Scheduling

In multi-tenant computing environments, users frequently submit jobs (both ML and non-ML) which can have differing priorities. To meet the stricter SLA requirements of high-priority jobs, a scheduler must sometimes re-allocate some resources used by a lower-priority job. If the lower-priority job is inelastic, then it may be killed or suspended, leaving the rest of its resources under-utilized and delaying its completion time. For long-running jobs such as training ML models, their resources may need to be re-allocated several times during their lifetimes.

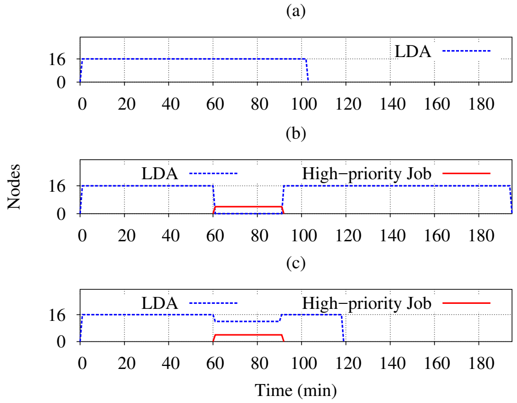


Figure 6: Priority scheduling experiments as described in Sec. 5.2.1. The graphs show the resource allocation over time in the cases of (a) LDA job which is uninterrupted, (b) LDA job which is killed when a higher-priority job is submitted, and (c) LDA job which elastically scales in when a higher-priority job is scheduled. We ran each experiment three times and saw negligible variation between each instance.

However, with the elasticity mechanism enabled by Litz, a long-running ML application can simply scale-in to use a fewer amount of resources, while the higher-priority job uses the released resources. After the higher-priority job completes, it can scale-out again, uninterrupted. We implemented this priority scheduling policy on a cluster of 16 m4.4xlarge nodes, and launched an LDA job on all 16 machines that runs for ≈100min if left uninterrupted (Fig. 6(a)). A higher-priority job is launched 60min into its runtime, requiring 4 nodes for 30min. Without elasticity, the LDA job is killed and re-started after the higher-priority job ends, requiring a total of ≈190min to complete (Fig. 6(b)). However, by leveraging elasticity to scale-in the LDA job, it can continue to run using 12 nodes and completes in ≈120min (Fig. 6(c)). At the same time, waiting for LDA to scale-in only increased the completion time of the high-priority job from 30min to 31min.

5.2.2 ML Resource Variability

The iterative-convergent nature of ML algorithms presents opportunities for resource scheduling not usually found in other computing tasks. One advantage of elasticity in an ML framework is that in addition to scaling in and out based on the directions from a cluster scheduler, an elastic ML framework can leverage resource variability that is inherent in ML applications to autonomously give up resources.

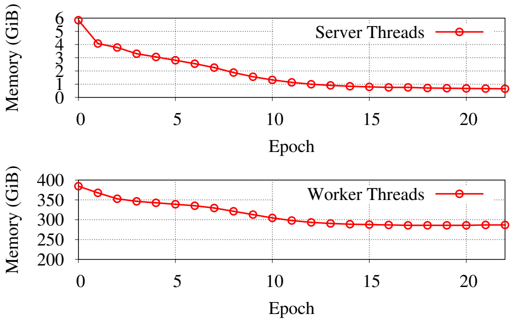


Figure 7: Memory usage on a cluster of 12 m4.4xlarge nodes during runtime of LDA implemented using Litz, broken down by server threads and worker threads. During the first 10 epochs, memory usage of server threads decrease by 5GiB, while memory usage of worker threads decrease by 70GiB.

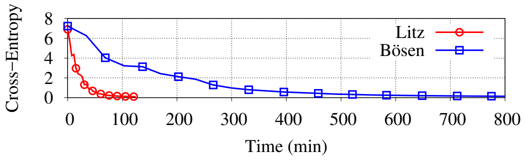


Figure 8: Multinomial Logistic Regression (MLR) running on 8 nodes using 25% of the ImageNet ILSVRC2012 dataset. Litz achieves convergence about 8× faster than Bösen.

In particular, many ML algorithms, including LDA, may find their model parameters becoming sparse (ie. mostly zeros) as they approach convergence [33], allowing memory usage to be reduced by using a more memory-efficient storage format (ie. sparse vector). Although LDA running on Strads has a similar decreasing memory usage, the lack of elasticity in Strads does not allow it to leverage this phenomenon for efficient scheduling.

Litz, on the other hand, can detect variability in the resource usage and reduce the number of worker and server threads accordingly. Fig. 7 shows the breakdown of memory usage during LDA. Server threads that store the model start with 6 GiB and drop to around 1 GiB by the 10th epoch, suggesting that the server threads can be reduced by 80%. Similarly, the worker threads start with 370 GiB of memory and reduce to about 300 GiB by the 10th epoch, suggesting that their count can be reduced by 20% and respective resources can be released. This dynamic resource usage of ML jobs, when exposed through an elastic framework like Litz, can inform the policies of a cluster scheduler that allocates resources between many jobs.

5.3 Performance Experiments

We compare our Litz implementations of MLR and LDA with those built-in with the open-source versions of Bösen and Strads, respectively. All three systems along with their applications are written using C++, and to further ensure fairness, we compiled all three using the `-O2 -g` flags and linked with the TCMalloc [21] memory allocator. These settings are the default for both Bösen and Strads.

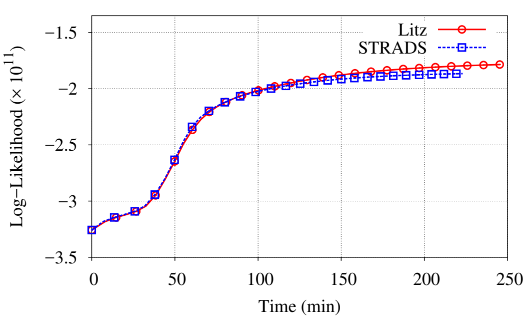


Figure 9: Latent Dirichlet Allocation (LDA) training algorithm running on Strads and Litz with the subsampled ClueWeb12 dataset. Litz completes all 34 epochs roughly 6% slower than Strads, but achieves a better objective value.

MLR Comparison with Bösen: We compare Litz with Bösen running the MLR application on 25% of the ImageNet ILSVRC2012 dataset[2] using 8 nodes. The open-source version of Bösen differs from the system described by Wei *et. al.* [50] in that it does not implement early communication nor update prioritization, but is otherwise the same and fully supports SSP execution. Both MLR instances were configured to use the same SSP staleness bound of 2 as well as the same SGD tuning parameters such as step size and minibatch size. As Fig. 8 shows, our MLR implementation on Litz converges about 8× faster than that on Bösen. Our profiling of Bösen and cursory examination of its code shows that it does not fully utilize CPUs due to lock contention. We believe the wide gap in performance is not due to fundamental architectural reasons, and that Bösen should be able to narrow the gap on such SSP applications given a more optimized implementation.

LDA Comparison with Strads: We next compare Litz with Strads running the LDA application using 12 nodes. The open-source version of Strads is the same implementation used in Kim *et. al.* [33]. Both LDA instances were configured to use the same number of block partitions as well as the same LDA hyper-parameters α and β. We ran each application until 34 epochs have been completed, where an *epoch* is equivalent to a full pass over the input data. As Fig. 9 shows, our LDA implementation on Litz completes all epochs roughly 6% slower than that on Strads. However, it also achieves a better objective value (measured in log-likelihood), resulting in faster convergence than Strads overall. Even though more investigation into the per-epoch convergence difference is needed, we can attribute the throughput difference to the optimizations built into Strads, which employs a ring-topology specifically optimized for the block-partitioned model scheduling strategy used by LDA.

Deep Neural Networks (DNNs): To evaluate Litz with DNNs, we implemented a particular deep learning model called a deep feed-forward network [22], which forms the

[2] With the full dataset, the Bösen baseline does not complete within a reasonable amount of time.

basis of many deep learning applications. We used a network with two hidden layers with ReLU activation and one output layer with Softmax activation. We trained this model using both Litz and TensorFlow [9] on 4 m4.4xlarge EC2 instances, with the CIFAR-10 [34] dataset. This dataset consists of 60K images, which are pre-processed into vectors of ≈98K features, labeled using 10 classes. Both systems used the same data-parallel SGD algorithm, and were configured with the same tuning parameters such as a learning rate of 0.0001 and mini-batch size of 64. The training using Tensorflow progressed at a pace of ≈79s per batch, while the training using Litz progressed 3.4× faster at a pace of ≈23s per batch.

6 Discussion and Related Work

Recently, there has been a growing interest in utilizing *transient* nodes in the cloud spot markets for big-data analytics. The systems developed for this setting try to execute jobs with the performance of *on-demand* nodes at a significantly cheaper cost, using transient nodes. The challenge for these systems is to deal with the bulk revocations efficiently by choosing right fault-tolerance mechanism. For example, SpotOn [47] dynamically determines the fault-tolerance mechanism that best balances the risk of revocation with the overhead of the mechanism. While SpotOn applies these fault-tolerance mechanisms at the systems level—using virtual machines or containers—Flint [46] argues that application-aware approach is preferable and can improve efficiency by adapting the fault-tolerance policy. Flint, which is based on Spark, proposes automated and selective checkpointing policies for RDDs, to bound the time Spark spends recomputing lost in-memory data after a bulk revocation of transient nodes. TR-Spark [54] argues that RDDs—the checkpointing unit in Spark—are too coarse-grained, making Spark unfit to run on transient resources, and takes Flint's approach further by providing fine-grained task-level checkpointing.

Unlike Flint and TR-Spark that adapt a general-purpose Spark framework to achieve cost-effective analytics with transient resources, Proteus [26] adapts a specialized ML framework to achieve significantly faster and cheaper execution, while introducing elasticity optimizations tuned for the setting. Specifically, Proteus stores the ML model on parameter servers that run on reliable on-demand nodes, and makes the workers stateless so that they can be run on transient node, effectively pushing workers' states to parameter servers, along with the model. This is a reasonable approach for the spot market setting where bulk revocations can take offline a large number of workers without notice. Although it works well for applications with small worker state, with an increasing data and model size, the approach may run into performance problems due to the communication overhead between workers and their state stored on the parameter servers. Litz, on the other hand, keeps the worker state in the workers and assumes a cooperative cluster scheduler that will ask the running application to give up nodes and wait for state to be transferred away. This approach results in high performance while still providing elasticity.

7 Conclusion and Future Work

We present the design and implementation of Litz, an evolutionary step in the elastic execution of ML applications in clouds and data-centers. We identify three important classes of distributed ML techniques—stateful workers, model scheduling, and relaxed consistency—and designed Litz's programming model to collectively support each of them. By adopting an event-driven API, Litz is able to control the execution of its applications, transparently migrating their state and computation between physical machines. Litz achieves elasticity—the ability to scale out and in based on changing resource availability—without compromising the state-of-the-art efficiency of non-elastic ML systems.

Furthermore, we describe the inherent dynamic memory usage of ML applications. We show that Litz is able to expose these patterns and significantly decrease its demand for memory across the lifetimes of ML jobs. Resource variability during the runtime of large data-analytics jobs is well known, and many schedulers have been introduced to exploit this variability for an efficient scheduling of jobs [32, 24, 23]. However, no previous work exists that exploit the specific resource usage patterns of ML applications. In future work, we plan to further investigate and identify the resource usage patterns of distributed ML applications, and then leverage their resource variability together with the elasticity of Litz for more efficient scheduling of ML jobs.

Lastly, we identify deep learning and elastic GPU computing as another interesting direction for future work. In particular, how does the relatively low-level event-driven API of Litz fit together with the higher-level symbolic programming models of deep learning frameworks like TensorFlow, MXNet [14], and DyNet [40]? With the current trend towards using compiler techniques to separate deep learning programming and execution [6, 7], we believe that frameworks like Litz will play an important role in the elastic and efficient execution of many future deep learning applications. The answers to these problems deserve thorough investigation.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We thank the members and companies of the PDL Consortium: Alibaba Group, Broadcom, Dell EMC, Facebook, Google, HP Enterprise, Hitachi, IBM Research, Intel, Micron, Microsoft Research, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate Technology, Two Sigma, Toshiba, Veritas and Western Digital for their interest, insights, feedback, and support. Our work was supported by the U.S. National Science Foundation awards IIS1447676 and CCF1629559, the Natural Sciences and Engineering Research Council of Canada award PGSD-471301-2015, as well as the Beijing Advanced Innovation Center for Big Data and Brain Computing at Beihang University.

References

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] Boost Context. www.boost.org/doc/libs/1_63_0/libs/context/.

[3] Boost Serialization. http://www.boost.org/doc/libs/1_64_0/libs/serialization/.

[4] etcd. http://coreos.com/etcd/.

[5] Kubernetes. http://kubernetes.io.

[6] NNVM. http://nnvm.tvmlang.org/.

[7] XLA. https://www.tensorflow.org/performance/xla/.

[8] ZeroMQ. http://zeromq.org.

[9] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning.

[10] AGARWAL, A., AND DUCHI, J. C. Distributed delayed stochastic optimization. In *Advances in Neural Information Processing Systems* (2011), pp. 873–881.

[11] AHAMAD, M., NEIGER, G., BURNS, J. E., KOHLI, P., AND HUTTO, P. W. Causal memory: definitions, implementation, and programming. *Distributed Computing 9*, 1 (Mar 1995), 37–49.

[12] AHN, S., SHAHBABA, B., WELLING, M., ET AL. Distributed stochastic gradient mcmc. In *ICML* (2014), pp. 1044–1052.

[13] BAILIS, P., VENKATARAMAN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND STOICA, I. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 776–787.

[14] CHEN, T., LI, M., LI, Y., LIN, M., WANG, N., WANG, M., XIAO, T., XU, B., ZHANG, C., AND ZHANG, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[15] CHILIMBI, T., SUZUE, Y., APACIBLE, J., AND KALYANARAMAN, K. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 571–582.

[16] CIPAR, J., HO, Q., KIM, J. K., LEE, S., GANGER, G. R., GIBSON, G., KEETON, K., AND XING, E. Solving the straggler problem with bounded staleness. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems* (Berkeley, CA, 2013), USENIX.

[17] DAI, W., KUMAR, A., WEI, J., HO, Q., GIBSON, G., AND XING, E. P. High-performance distributed ml at scale through parameter server consistency models. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (2015), AAAI'15, AAAI Press, pp. 79–87.

[18] DEAN, J., CORRADO, G., MONGA, R., CHEN, K., DEVIN, M., MAO, M., SENIOR, A., TUCKER, P., YANG, K., LE, Q. V., ET AL. Large scale distributed deep networks. In *Advances in neural information processing systems* (2012), pp. 1223–1231.

[19] GABRILOVICH, E., RINGGAARD, M., AND SUBRAMANYA, A. Facc1: Freebase annotation of clueweb corpora, version 1 (release date 2013-06-26, format version 1, correction level 0). http://lemurproject.org/clueweb12/, 2013.

[20] GEMULLA, R., NIJKAMP, E., HAAS, P. J., AND SISMANIS, Y. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2011), KDD '11, ACM, pp. 69–77.

[21] GHEMAWAT, S., AND MENAGE, P. TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[22] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[23] GRANDL, R., CHOWDHURY, M., AKELLA, A., AND ANANTHANARAYANAN, G. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 65–80.

[24] GRANDL, R., KANDULA, S., RAO, S., AKELLA, A., AND KULKARNI, J. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 81–97.

[25] HARLAP, A., CUI, H., DAI, W., WEI, J., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Addressing the straggler problem for iterative convergent parallel ml. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 98–111.

[26] HARLAP, A., TUMANOV, A., CHUNG, A., GANGER, G., AND GIBBONS, P. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM.

[27] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 295–308.

[28] HO, Q., CIPAR, J., CUI, H., LEE, S., KIM, J. K., GIBBONS, P. B., GIBSON, G. A., GANGER, G., AND XING, E. P. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems 26*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 1223–1231.

[29] HONG, M. A distributed, asynchronous and incremental algorithm for nonconvex optimization: An admm based approach. *arXiv preprint arXiv:1412.6058* (2014).

[30] HUANG, Q., SU, S., XU, S., LI, J., XU, P., AND SHUANG, K. Migration-based elastic consolidation scheduling in cloud data center. In *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops* (July 2013), pp. 93–97.

[31] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 11–11.

[32] JYOTHI, S. A., CURINO, C., MENACHE, I., NARAYANAMURTHY, S. M., TUMANOV, A., YANIV, J., MAVLYUTOV, R., GOIRI, I., KRISHNAN, S., KULKARNI, J., AND RAO, S. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 117–134.

[33] KIM, J. K., HO, Q., LEE, S., ZHENG, X., DAI, W., GIBSON, G. A., AND XING, E. P. Strads: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 5:1–5:16.

[34] KRIZHEVSKY, A. Learning multiple layers of features from tiny images.

[35] KUMAR, A., BEUTEL, A., HO, Q., AND XING, E. P. Fugue: Slow-worker-agnostic distributed learning for big models on big data. In *AISTATS* (2014), pp. 531–539.

[36] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 583–598.

[37] LI, M., ANDERSEN, D. G., AND SMOLA, A. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning* (2013).

[38] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 716–727.

[39] MCMAHAN, B., AND STREETER, M. Delay-tolerant algorithms for asynchronous distributed online learning. In *Advances in Neural Information Processing Systems* (2014), pp. 2915–2923.

[40] NEUBIG, G., DYER, C., GOLDBERG, Y., MATTHEWS, A., AMMAR, W., ANASTASOPOULOS, A., BALLESTEROS, M., CHIANG, D., CLOTHIAUX, D., COHN, T., ET AL. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980* (2017).

[41] PUNDIR, M., KUMAR, M., LESLIE, L. M., GUPTA, I., AND CAMPBELL, R. H. Supporting on-demand elasticity in distributed graph processing. In *Cloud Engineering (IC2E), 2016 IEEE International Conference on* (2016), IEEE, pp. 12–21.

[42] RECHT, B., RE, C., WRIGHT, S., AND NIU, F. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 693–701.

[43] RUSSAKOVSKY, O., DENG, J., SU, H., KRAUSE, J., SATHEESH, S., MA, S., HUANG, Z., KARPATHY, A., KHOSLA, A., BERNSTEIN, M., BERG, A. C., AND FEI-FEI, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV) 115*, 3 (2015), 211–252.

[44] SCHERRER, C., TEWARI, A., HALAPPANAVAR, M., AND HAGLIN, D. Feature clustering for accelerating parallel coordinate descent. In *Advances in Neural Information Processing Systems* (2012), pp. 28–36.

[45] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)* (Prague, Czech Republic, 2013), pp. 351–364.

[46] SHARMA, P., GUO, T., HE, X., IRWIN, D., AND SHENOY, P. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 6:1–6:15.

[47] SUBRAMANYA, S., GUO, T., SHARMA, P., IRWIN, D., AND SHENOY, P. Spoton: A batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 329–341.

[48] WANG, J., YANG, J., YU, K., LV, F., HUANG, T., AND GONG, Y. Locality-constrained linear coding for image classification. In *IN: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN CLASSIFICATOIN* (2010).

[49] WANG, M., XIAO, T., LI, J., ZHANG, J., HONG, C., AND ZHANG, Z. Minerva: A scalable and highly efficient training platform for deep learning. In *NIPS Workshop, Distributed Machine Learning and Matrix Computations* (2014).

[50] WEI, J., DAI, W., QIAO, A., HO, Q., CUI, H., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 381–394.

[51] WEI, J., KIM, J. K., AND GIBSON, G. A. Benchmarking Apache Spark with Machine Learning Applications. In *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-16-107, Oct. 2016.*

[52] XIE, P., KIM, J. K., ZHOU, Y., HO, Q., KUMAR, A., YU, Y., AND XING, E. P. Distributed machine learning via sufficient factor broadcasting. *CoRR abs/1511.08486* (2015).

[53] XING, E. P., HO, Q., DAI, W., KIM, J. K., WEI, J., LEE, S., ZHENG, X., XIE, P., KUMAR, A., AND YU, Y. Petuum: A new platform for distributed machine learning on big data. *IEEE Trans. Big Data 1*, 2 (2015), 49–67.

[54] YAN, Y., GAO, Y., CHEN, Y., GUO, Z., CHEN, B., AND MOSCIBRODA, T. Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 484–496.

[55] YUAN, J., GAO, F., HO, Q., DAI, W., WEI, J., ZHENG, X., XING, E. P., LIU, T.-Y., AND MA, W.-Y. Lightlda: Big topic models on modest computer clusters. In *Proceedings of the 24th International Conference on World Wide Web* (2015), ACM, pp. 1351–1361.

[56] YUN, H., YU, H.-F., HSIEH, C.-J., VISHWANATHAN, S., AND DHILLON, I. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *Proceedings of the VLDB Endowment 7*, 11 (2014), 975–986.

[57] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.

[58] ZHANG, R., AND KWOK, J. T. Asynchronous distributed admm for consensus optimization. In *ICML* (2014), pp. 1701–1709.

Putting the “Micro” Back in Microservice

Sol Boucher*, Anuj Kalia*, David G. Andersen*, and Michael Kaminsky[†]
* *Carnegie Mellon University* [†] *Intel Labs*

Abstract

Modern cloud computing environments strive to provide users with fine-grained scheduling and accounting, as well as seamless scalability. The most recent face to this trend is the “serverless” model, in which individual functions, or microservices, are executed on demand. Popular implementations of this model, however, operate at a relatively coarse granularity, occupying resources for minutes at a time and requiring hundreds of milliseconds for a cold launch. In this paper, we describe a novel design for providing “functions as a service” (FaaS) that attempts to be truly *micro*: cold launch times in microseconds that enable even finer-grained resource accounting and support latency-critical applications. Our proposal is to eschew much of the traditional serverless infrastructure in favor of language-based isolation. The result is microsecond-granularity launch latency, and microsecond-scale preemptive scheduling using high-precision timers.

1 Introduction

As the scope and scale of Internet services continues to grow, system designers have sought platforms that simplify scaling and deployment. Services that outgrew self-hosted servers moved to datacenter racks, then eventually to virtualized cloud hosting environments. However, this model only partially delivered two related benefits:

1. Pay for only what you use at very fine granularity
2. Scale up rapidly on demand

The VM approach suffered from relatively coarse granularity: Its atomic compute unit of machines were billed at a minimum of minutes to months. Relatively long startup times often required system designers to keep some spare capacity online to handle load spikes.

These shortcomings led cloud providers to introduce a new model, known as serverless computing, in which the customer provides *only* their code, without having to configure its environment. Such “function as a service” (FaaS) platforms are now available as AWS Lambda [4], Google Cloud Functions [10], Azure Functions [18], and Apache OpenWhisk [5]. These platforms provide a model in which: (1) user code is invoked whenever some event occurs (e.g., an HTTP API request), runs to completion, and nominally stops running (and being billed) after it completes; and (2) there is no state preserved between separate invocations of the user code. Property (2) enables easy auto-scaling of the function as load changes.

Because these services execute within a cloud provider’s infrastructure, they benefit from low-latency access to other cloud services. In fact, acting as an access-control proxy is a recurring microservice pattern: receive an API request from a user, validate it, then access a backend storage service (e.g., S3) using the service’s credentials.

In this paper, we explore a design intended to reduce the tension between two of the desiderata for cloud functions: low latency invocation and low cost. Contemporary invocation techniques exhibit high latency with a large tail; this is unsuitable for many modern distributed systems which involve high-fanout communication, sometimes performing thousands of lookups to handle each user request. Because user-visible response time often depends on the tail latency of the slowest chain of dependent responses [7], shrinking the tail is crucial [11, 24, 16, 12].

Thus we seek to reduce the invocation latency and improve predictability, a goal supported by the impressively low network latencies available in modern datacenters. For example, it now takes $< 20\mu s$ to perform an RPC between two machines in Microsoft Azure’s virtual machines [9]. We believe, however, that fully leveraging this improving network performance will require reducing microservices’ invocation latencies to the point where the network is once again the bottleneck.

We further hypothesize—admittedly without much proof for this chicken-and-egg scenario—that substantially reducing both the latency and cost of running intermittently-used services will enable new classes and scales of applications for cloud functions, and in the remainder of this paper, present a design that achieves this. As Lampson noted, there is power in making systems “fast rather than general or powerful” [14], because fast building blocks can be used more widely.

Of course, a microservice is only as fast as the slowest service it relies on; however, recall that many such services are offered in the same clouds and datacenters as serverless platforms. Decreasing network latencies will push these services to respond faster as well, and new stable storage technologies such as 3D XPoint (projected to offer sub-microsecond reads and writes) will further accelerate this trend by offering lower-latency storage.

In this paper, we propose a restructuring of the serverless model centered around low-latency: *lightweight microservices* run in *shared processes* and are isolated primarily with language-based *compile-time guarantees* and *fine-grained preemption*.

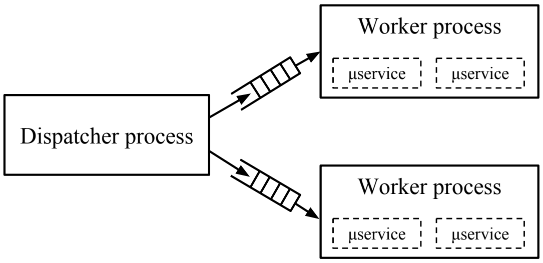


Figure 1: Language-based isolation design. The dispatcher process uses shared in-memory queues to feed requests to the worker processes, each of which runs one user-supplied microservice at a time.

2 Motivation

Our decision to use language-based isolation is based on two experimental findings: (1) Process-level isolation is too slow for microsecond-scale user functions. (2) Commodity CPUs support task preemption at microsecond scale. We conducted our experiments on an Intel® Xeon® E5-2683 v4 server (16 cores, 2.1 GHz) and Linux 4.13.0.[1]

2.1 Process-level isolation is too slow

We use a single-machine experiment to evaluate the invocation overhead of different isolation mechanisms: Microservices run on 14 *worker* CPU cores. Another core runs a *dispatcher* process that launches microservices on the workers. All requests originate at the dispatcher (which in a full serverless platform would forward from a cluster scheduler); it schedules ≤14 microservices at a time, one per worker core, choosing from a pool of 5,000.

To provide a comparison against contemporary system designs, we use two different isolation mechanisms:

1. **Process-based isolation:** Each microservice is a separate process. We expect this approach to exhibit latency at least as low as the container isolation common in contemporary serverless deployments.
2. **Language-based isolation:** Each worker core hosts a single-threaded *worker process* that directly executes different microservices, one at a time. In this approach, shown in Figure 1, a worker process runs a microservice by calling its registered function; we assume that the microservice function can be isolated from the worker process with language-based isolation techniques that we discuss in Section 3. The dispatcher schedules microservices on worker processes by sending them requests on a shared memory queue, which idle worker processes poll.

We use 5,000 copies of a Rust microservice that simply records a timestamp: latency is measured between when the dispatcher invokes a microservice and the time that microservice records. There are two experiment modes:

Microservices		Latency (µs)		Throughput
Resident?	**Isolation**	**Median**	**99%**	**(M invoc/s)**
Warm-start	Process	8.7	27.3	0.29
	Language	1.2	2.0	5.4
Cold-start	Process	2845.8	15976.0	–
	Language	38.7	42.2	–

Table 1: Microservice invocation performance

Warm-start requests. We first model a situation where all of the microservices are already resident on the compute node. In the case of process-based isolation, the dispatcher launches all 5,000 microservices at the beginning of the experiment, but they all block on an IPC call; the dispatcher then invokes each microservice by waking up its process using a UDP datagram. In the case of language-based isolation, the microservices are dynamic libraries preloaded into the worker processes.

Table 1 shows the latency and throughput of the two methods. We find that the process-based isolation approach takes 9 µs and achieves only 300,000 warm microservice invocations per second. In contrast, language-based isolation achieves 1.2 µs latency (with a tail of just 2.0 µs) and over 5 million invocations per second.

Considering that the FaRM distributed computing platform achieved mean TATP transaction commit latencies as low as 19 µs in 2015 [8], a 9 µs microservice invocation delay represents almost 50% overhead for a microservice providing a thin API gateway to such a backend. We therefore conclude that even in the average case, process-based isolation is too slow for microsecond-scale scheduling. Furthermore, IPC overhead limits invocation throughput.

Process-based isolation also has a higher memory footprint: loading the 5,000 trivial microservices consumes 2 GiB of memory with the process-based approach, but only 1.3 GiB with the language-based one. However, this benefit may reduce as microservices' code sizes increase.

Cold-start requests. Achieving ideal wakeup times is possible only when the microservices are already resident, but the tail latency of the serverless platform depends on those requests whose microservices must be loaded before they can be invoked. To assess the difference between process-based and language-based isolation in this context, we run the experiment with the following change: In the former case, the dispatcher now launches a transient microservice process for each request by `fork()/exec()`'ing. In the latter, the dispatcher asks a worker to load a microservice's dynamic library (and unload it afterward). The results in Table 1 reveal an order-of-magnitude slip in the language-based approach's latency; however, this is overshadowed by the three orders of magnitude increase for process-based isolation.

[1] Source code for the benchmarks in this paper is available from https://github.com/efficient/microservices_microbenchmarks.

2.2 Intra-process preemption is fast

In a complete serverless platform, some cluser-level scheduler would route incoming requests to individual worker nodes. Since we run user-provided microservices directly in worker processes, a rogue long-running microservice could thwart such scheduling by unexpectedly consuming the resources of a worker that already had numerous other requests queued. We hypothesize that, in such situations, it is better for tail latency to preempt the long microservice than retarget the waiting jobs to other nodes in real time. (Only the compute node already assigned a request is well positioned to know whether that request is being excessively delayed: whereas other nodes can only tell that the request hasn't yet *completed*, this node alone knows whether it has been *scheduled*.) At our scale, this means a preemption interval up to two orders of magnitude faster than Linux's default 4 ms process scheduling quantum.

Fortunately, we find that high-precision event timers (HPETs) on modern CPUs are sufficient for this task. We measure the granularity and reliability of these timers as follows: We install a signal handler and configure a POSIX timer to trigger it every T μs. Ideally, this handler would always be called exactly T μs after its last invocation; we measure the deviation from T over 65,535 iterations. We find that the variance is smaller than 0.5 μs for $T \geq 3$ μs. This shows that intra-process preemption is fast and reliable enough for our needs.

3 Providing Isolation

Consolidating multiple users' jobs into a single process requires addressing security and isolation. We aim to do it without compromising our ambitious performance goals.

Our guiding philosophy for achieving this is "language-based isolation with defense in depth." We draw inspiration from two recently-published systems whose own demanding performance requirements drove them to perform similar coalescing of traditionally independent components: NetBricks [19] is a network functions runtime for providing programmable network capabilities; it is unique among this class of systems for running third-party network functions in-process rather than in VMs. Tock [15] is an embedded microkernel whose servers ("capsules") form a common compilation unit and communicate using type-safe function calls. As their primary defense against untrusted code, both systems leverage Rust [3], a new type-safe systems programming language.

Rust is a strongly-typed, compiled language that uses a lightweight runtime similar to C. Unlike many other modern systems languages, Rust is an attractive choice for predictable performance because it does not use a garbage collector. It provides strong memory safety guarantees by focusing on "zero-cost abstractions" (i.e., those that can be compiled down to code whose safety is assured without runtime checks). In particular, safe Rust code is guaranteed to be free of null or dangling pointer dereferences, invalid variable values (e.g., casts are checked and unions are tagged), reads from uninitialized memory, mutations of non-`mut` data (roughly the equivalent of C's `const`), and data races, among other misbehaviors [22].

We require each microservice to be written in Rust (although, in the future, it might be possible to support subsets of other languages by compiling them to safe Rust), giving us many aspects of the isolation we need. It is difficult for microservices to crash the worker process, since most segmentation faults are prevented, and runtime errors such as integer overflow generate Rust panics that we can catch. Microservices cannot get references to data that does not belong to them thanks to the variable and pointer initialization rules.

Given our performance goals, there is a crucial isolation aspect that Rust does not provide: there is nothing to stop users from monopolizing the CPU. Our system, however, must be preemptive. We are unaware of existing preemption techniques that work at microsecond scales. Note that coroutine-like cooperative multitasking approaches (such as lightweight threads in Go [2] and Erlang [1]) are not preemptive, so they do not work for us. We briefly discuss our solution to this in the following section; it depends on installing a `SIGALRM` handler and ensuring that trusted code within the process handles the signal.

Our defense-in-depth comes from using lightweight operating-system protections to block access to certain system calls, as well as the proposed mechanisms in Section 6. Some system calls must be blocked to have any defense at all; otherwise, the microservice could create kernel threads (e.g., `fork()`), create competition between threads (e.g., `nice()`), or even terminate the entire worker (e.g., `exit()`). Finally, user functions should not have unmonitored file system access.

We propose to block system calls using Linux's `seccomp()` system call [20]; each worker process should call this during initialization to limit itself to a whitelisted set of system calls. Prior to lockdown, the worker process should install a `SIGSYS` handler for regaining control from any microservice that attempts to violate the policy.

4 Providing Preemption

The system must be able to detect and recover from microservices that, whether maliciously or negligently, attempt to run for longer than permitted. The two parts of this problem are (1) regaining control of the CPU and (2) aborting and cleaning up after the user code.

As proposed in Section 2, regaining control of the CPU happens when a signal (`SIGALRM`) from the kernel transfers control to the worker process's handler.[2] The handler then checks how long the current microservice

[2] For defense in depth, the worker process should be prevented from subsequently modifying this signal-handling configuration.

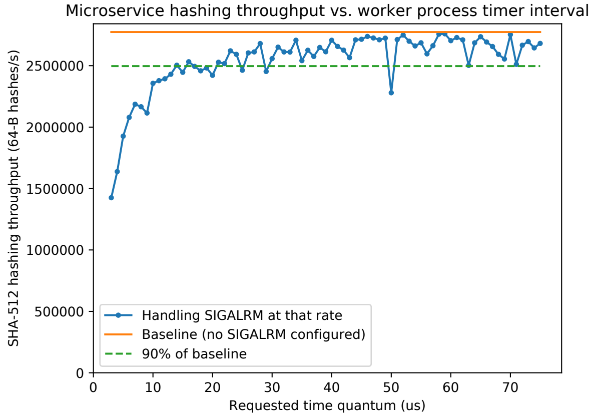


Figure 2: Effect of `SIGALRM` quantum on hashing tput.

has been running and decides whether it should be killed. (We register the handler using the `SA_RESTART` flag to `sigaction()` so that any interrupted blocking syscalls are restarted transparently.) However, there remain three important questions:

For how long should each microservice be allowed to run? Assume that each core executes one user task at a time and that all microservice functions are pre-compiled and resident (warm invocation). We define L to be the desired warm invocation latency, B to be the runtime budget allotted to each microservice, and r_c to be the remaining runtime of the microservice on CPU c. Thus, in the worst case (where all tasks are executing for their entire allotted time) the probability that the incoming microservice will have somewhere to run in time to meet the invocation latency SLO is given by:

$$p(r_{\text{min}} \leq L) = \sum_{c \in C} p(r_c \leq L) = |C| \frac{L}{B} \qquad (1)$$

Given the 14 cores in our setup and imagining we want to keep the 99% tail, $p(r_{\text{min}} \leq L) = 0.99$, to an L of 8 μs, we need to kill tasks running for more than B = 113 μs.

How often should the handler execute (the *quantum*)? We showed in Section 2 that microsecond-scale preemption is *achievable*, but can it be done *efficiently*? To find out, we wrote a microservice that measures the throughput of computing SHA-512 hashes over 64 B of data at a time. We then subjected its worker process to `SIGALRM`s, varying the quantum and observing the resulting hashing throughput. Figure 2 illustrates that by a quantum of about 20 µs, throughput had reached around 90% of baseline. Considering this performance degradation, acceptable we adopt this quantum and prescribe a runtime budget of 113 - 20 = 93 µs so that we can kill over-budget microservices in time to avoid violating our tail latency SLO.

How do we clean up a terminated microservice? We now discuss our mechanism for aborting and cleaning up after a microservice exceeds its runtime budget. POSIX signal handlers receive as an argument a pointer to their *context*, a snapshot of the process's PCB (process control block) at the moment before it received the signal. When the handler returns, the system will transfer control back to the point described by the context, so a naïve way for our worker processes to regain control would be to reset its GPRs (general-purpose registers) to values recorded just before the worker's tight scheduling loop. This approach, however, would not release the microservice's state or memory allocations back to the worker.

One of the few heavyweight components of the Rust runtime is panic handling, reminiscent of C++'s exception handling. The compiler inserts landing pads into each function that call the destructors for the variables in its stack frame: if the program ever panics, the standard library uses these to unwind the stack. We co-opt this functionality by having the `SIGALRM` handler set its context to raise an explicit panic in a fake stack frame just above the real top of the stack.

Section 6 discusses the limitations and security ramifications of this approach.

5 Deployment

We now describe our microservices in the broader context of our full proposed serverless system. We clarify their lifecycle, interactions with the compute nodes, and the trust model for the cloud provider.

Users submit their microservices in the form of Rust source code, allowing the serverless operator to pass the `-Funsafe-code` compilation flag to reject any `unsafe` code. This process need not occur on the compute nodes, provided the deployment server tasked with compilation runs the same version of the Rust compiler.[3] The operator needs to trust the compiler, standard library, and any libraries against which it will permit the microservice to link (since they might contain `unsafe` code), but importantly need not worry about the microservice itself.

We believe that restricting microservices to a specific list of permitted dependencies is reasonable. Any library that contains only safe Rust code could be whitelisted without review. To approximate the size of such a list given the current Rust ecosystem, we turn to a 2017 study [6] by the Tock authors that found just under half of the Rust package manager's top 1000 most-downloaded libraries to be free of unsafe code. They caution that many of those packages have unsafe dependencies, but reviewing a relatively small number of popular libraries would open up the majority of the most popular packages.

If the application compiles (is proven memory-safe) and links (depends only on trusted libraries) successfully, the deployment server produces a shared object file, which the provider then distributes to each compute node on which it might run. Then, in order to ensure that invokers will experience the warm-start latencies discussed in Section 2,

[3] This restriction exists because, as of the latest release (1.23.0) of the compiler, Rust does not have a stable ABI.

those nodes' dispatcher processes should instruct one or more of their workers to preload the dynamic library. If the provider experiences too many active microservices for its available resources, it can unload some libraries; on their next invocation, they will experience higher (cold start) invocation latencies as they synchronously load the dynamic library.

6 Future Work

As noted above, our exploration is preliminary; this section outlines several open questions. These questions fall into two categories: shortcomings in our current implementation and defense-in-depth safeguards against unexpected failures (e.g., compiler bug or the operator allowing use of a buggy or malicious library).

Non-reentrancy. Our use of Rust panics to unwind the stack during preemption can corrupt the internal state of non-reentrant functions (e.g., Rust's dynamic allocator). Possible fixes include blacklisting these functions and delaying preemption until they are finished or replacing the problematic function with a safe one (e.g., a custom memory allocator).

Host process. Our current implementation does not provide isolation between the dispatcher and worker processes. We plan to apply standard OS techniques to reduce the chance of interference by a misbehaving worker. Examples include auditing interactions with the shared memory region to ensure invalid or inconsistent data originating from a worker cannot create an unrecoverable dispatcher error; handling the `SIGCHLD` signal to detect a worker that has somehow crashed; and keeping a recovery log in the dispatcher process so that any user jobs lost to a failed worker process can be reassigned to operational workers.

Further defense in depth with ERIM. ERIM outlines a set of techniques and binary rewriting tools useful for using Intel's Memory Protection Keys to restrict memory access by threads within a process [23]. While preliminary and without source yet available, this appears to be an attractive approach for defense-in-depth both within worker processes and between the workers and the dispatcher.

Library functions. As with system calls, there may exist library functions in Rust (and certainly in libc, which we deny by default) that are unsafe for microservices to access. Because the Rust standard library requires unsafe code, defense-in-depth suggests that a whitelisting-based approach should be employed for access to its functions. Certainly library functions must be masked—for example, our use of Rust's panic handler for preemption means that we must deny microservice code the ability to catch the panic and return to execution. Although we mitigate this possibility by detecting and blacklisting microservices that fail to yield under a SIGALRM, it would be desirable to block such behavior entirely. Possible options include using the dynamic linker to interpose stub implementations or linking against a custom build of the library, or using more in-depth static analysis.

Resource leaks. Safe Rust code provides memory safety, but it cannot prevent memory leaks [21]. For example, destructor invocation is not guaranteed using Rust's default reference counting-based reclamation; therefore, unwinding the stack during preemption is not guaranteed to free all of a microservice's memory or other resources. Potential solutions are interposing on the dynamic allocator to record tracking information (likely proving expensive) or using per-microservice heaps that main worker process can simply deallocate when terminating a microservice. The worker can also deallocate other resources, such as unclosed file descriptors. If these checks end up being too expensive, the worker could execute its cleanup after a certain number of microservices have run or when the load is sufficiently low.

Side channels. Our current approach is vulnerable to side-channel attacks [17, 13]. For example, microservices have access to the memory addresses and timings of dynamic memory allocations, as well as the numbers of opened file descriptors. Although side-channels exist in many systems, the short duration of microservice functions may make mounting such attacks more challenging; nevertheless, standard preventative practices found in the literature should apply.

Despite the security challenges of running microservice as functions, worker processes are still well-isolated from the rest of the system. Worst case, the central dispatcher process can restart a failed worker and automatically ban suspect microservices.

7 Conclusion

In order to permit applications to fully leverage the 10s of μs latencies available from the latest datacenter networks, we propose a novel design for serverless platforms that runs user-submitted microservices within shared processes. This structure is possible because of language-based *compile-time memory safety guarantees* and *microsecond-scale preemption*. Our implementation and experiments demonstrate that these goals of high throughput, low invocation latency, and rapid preemption are achievable on today's commodity systems, while potentially supporting hundreds of thousands of concurrently available microservices on each compute node. We believe that these two building blocks will enable new FaaS platforms that can deliver single-digit microsecond invocation latencies for lightweight, short-lived tasks.

Acknowledgements

This work was supported by the U.S. National Science Foundation under award CCF-1535821 and the Intel Science and Technology Center for Visual Cloud Systems.

References

[1] Erlang programming language. https://www.erlang.org, 2018.

[2] The Go programming language. https://golang.org, 2018.

[3] The Rust programming language. https://www.rust-lang.org, 2018.

[4] Amazon. AWS Lambda. https://aws.amazon.com/lambda.

[5] Apache Software Foundation. OpenWhisk. https://openwhisk.apache.org.

[6] Brad Campbell. Crates.io ecosystem not ready for embedded Rust. https://www.tockos.org/blog/2017/crates-are-not-safe.

[7] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[8] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[9] D. Firestone et al. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, Apr. 2018.

[10] Google. Cloud Functions. https://cloud.google.com/functions.

[11] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[12] M. Jeon, Y. He, H. Kim, S. Elnikety, S. Rixner, and A. L. Cox. TPC: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[13] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[14] B. W. Lampson. Hints for computer system design. In *Proceedings of the ninth ACM symposium on operating systems principles*, SOSP '83, pages 33–48. ACM, 1983.

[15] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64 kB computer safely and efficiently. In *Proceedings of the 26th ACM symposium on operating systems principles*, SOSP '17, pages 234–251. ACM, 2017.

[16] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[17] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[18] Microsoft. Azure Functions. https://azure.microsoft.com/services/functions.

[19] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.

[20] seccomp(). seccomp(2) manual page from Linux man-pages project, Nov. 2017.

[21] The Rust Reference. Behavior not considered unsafe. https://doc.rust-lang.org/stable/reference/behavior-not-considered-unsafe.html, 2018.

[22] The Rust Reference. Behavior considered undefined. https://doc.rust-lang.org/stable/reference/behavior-considered-undefined.html, 2018.

[23] A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, and P. Druschel. Erim: Secure and efficient in-process isolation with memory protection keys. *arXiv preprint arXiv:1801.06822*, 2018.

[24] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

Fast and Concurrent RDF Queries using RDMA-assisted GPU Graph Exploration

Siyuan Wang, Chang Lou, Rong Chen, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Contacts: {rongchen, haibochen}@sjtu.edu.cn

Abstract

RDF graph has been increasingly used to store and represent information shared over the Web, including social graphs and knowledge bases. With the increasing scale of RDF graphs and the concurrency level of SPARQL queries, current RDF systems are confronted with inefficient concurrent query processing on massive data parallelism, which usually leads to suboptimal response time (latency) as well as throughput.

In this paper, we present Wukong+G, the first graph-based distributed RDF query processing system that efficiently exploits the hybrid parallelism of CPU and GPU. Wukong+G is made fast and concurrent with three key designs. First, Wukong+G utilizes GPU to tame random memory accesses in graph exploration by efficiently mapping data between CPU and GPU for latency hiding, including a set of techniques like query-aware prefetching, pattern-aware pipelining and fine-grained swapping. Second, Wukong+G scales up by introducing a GPU-friendly RDF store to support RDF graphs exceeding GPU memory size, by using techniques like predicate-based grouping, pairwise caching and look-ahead replacing to narrow the gap between host and device memory scale. Third, Wukong+G scales out through a communication layer that decouples the transferring process for query metadata and intermediate results, and leverages both native and GPUDirect RDMA to enable efficient communication on a CPU/GPU cluster.

We have implemented Wukong+G by extending a state-of-the-art distributed RDF store (i.e., Wukong) with distributed GPU support. Evaluation on a 5-node CPU/GPU cluster (10 GPU cards) with RDMA-capable network shows that Wukong+G outperforms Wukong by 2.3X-9.0X in the single heavy query latency and improves latency and throughput by more than one order of magnitude when facing hybrid workloads.

1 Introduction

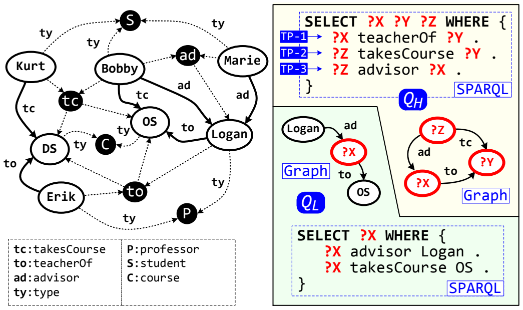
Resource Description Framework (RDF) is a standard data model for the Semantic Web, recommended by W3C [5]. RDF describes linked data as a set of triples forming a highly connected graph, which powers information retrievable through the query language SPARQL. RDF and SPARQL have been widely used in Google's knowledge graph [22] and many public knowledge bases, such as DBpedia [1], PubChemRDF [38], Wikidata [8], Probase [59], and Bio2RDF [10].

The drastically increasing scale of RDF graphs has posed a grand challenge to fast and concurrent queries over large RDF datasets [17]. Currently, there have been a number of systems built upon relational databases, including both centralized [40, 12, 58] and distributed [48, 44, 23] designs. On the other hand, Trinity.RDF [62] uses graph exploration to reduce the costly join operations in intermediate steps but still requires a final join operation. To further accelerate distributed query processing, Wukong [51] leverages RDMA-based graph exploration to support massively concurrency queries with low latency requirement and adopts full-history pruning to avoid the final join operation.

Essentially, many RDF queries have embarrassing parallelism, especially for *heavy* queries, which usually touch a large portion of the RDF graph on an excessive amount of paths using graph exploration. This poses a significant challenge even for multicore CPUs to handle them efficiently, which usually causes lengthy execution time. For example, the latency differences among seven queries in LUBM [7] is more than 3,000X (0.13ms and 390ms for Q5 and Q7 accordingly). This may cause one heavy query block all other queries, substantially extending the latency of other queries and dramatically impairing the throughput of processing concurrent queries [51]. This problem has also gained increased attention [45].

In this paper, we present Wukong+G[1] with a novel design that exploits a distributed heterogeneous CPU/GPU cluster to accelerate heterogeneous RDF queries based on distributed graph exploration. Unlike CPUs pursuing the minimized execution time for single instructions, GPUs are designed to provide high computational throughput for massive simple control-flow operations with little or no control dependency. Such features expose a design space to distribute hybrid workloads by offloading heavy queries to GPUs. Nevertheless, different from many traditional GPU workloads, RDF graph queries are memory-intensive instead of compute-intensive: there are limited arithmetic operations and most of the processing time is spent on random memory accesses. This unique feature implies that the key of performance optimizations in Wukong+G is on *smart*

[1] The source code and a brief instruction of Wukong+G are available at http://ipads.se.sjtu.edu.cn/projects/wukong.



```
SELECT ?X ?Y ?Z WHERE {
  ?X teacherOf ?Y .
  ?Z takesCourse ?Y .
  ?Z advisor ?X .
}
```

```
SELECT ?X WHERE {
   ?X advisor Logan .
   ?X takesCourse OS .
}
```

Fig. 1: A sample of RDF data and two SPARQL queries (Q_H and Q_L). White circles indicate the normal vertices (*subjects* and *objects*); dark circles indicate the (*type* and *predicate*) index vertices. Q_H is a *heavy* query, and Q_L is a *light* query.

memory usage rather than improving the computation algorithm. Wukong+G is made fast and concurrent with the following key designs:

GPU-based query execution (§4.1). To achieve the best performance for massive random accesses demanded by heavy queries, Wukong+G leverages the many-core feature and latency hiding ability of GPUs. Besides making use of hardware advantages, Wukong+G surmounts the limitations of GPU memory size and PCIe (PCI Express) bandwidth by adopting *query-aware prefetching* to mitigate the constraints on graph size, *pattern-aware pipelining* to hide data movement cost, and *fine-grained swapping* to minimize data transfer size.

GPU-friendly RDF store (§4.2). To support desired CPU/GPU co-execution pattern while still enjoying the fast graph exploration, Wukong+G follows a distributed in-memory key/value store and proposes a *predicate-based grouping* to aggregate keys and values with the same predicate individually. Wukong+G further smartly manages GPU memory as a cache of RDF store by supporting *pairwise caching* and *look-ahead replacing*.

Heterogeneous RDMA communication (§4.3). To preserve better communication efficiency in a heterogeneous environment, Wukong+G decouples the transferring process of query metadata and intermediate results for SPARQL queries. Wukong+G uses native RDMA to send metadata like query plan and current step among CPUs, and uses GPUDirect RDMA to send current intermediate results (history table) directly among GPUs. This preserves the performance boost brought by GPUs from potential expensive CPU/GPU data transfer cost.

We have implemented Wukong+G by extending Wukong [51], a state-of-the-art distributed RDF query system to support heterogeneous CPU/GPU processing. To confirm the performance benefit of Wukong+G, we have conducted a set of evaluations on a 5-node CPU/GPU cluster (10 GPU cards) with RDMA-capable network. The experimental results using the LUBM [7] benchmark show that Wukong+G outperforms Wukong by 2.3X-9.0X in the single heavy query latency and improves latency and throughput by more than one order of magnitude when facing hybrid workloads.

2 Background and Motivation

2.1 RDF and SPARQL

An RDF dataset is composed by triples, in the form of $\langle subject, predicate, object \rangle$. To construct a graph (aka RDF graph), each triple can be regarded as a directed edge (*predicate*) connecting two vertices (from *subject* to *object*). In Fig. 1, a simplified sample RDF graph of LUBM dataset [7] includes two professors (Logan and Erik), three students (Marie, Bobby, and Kurt), and two courses (OS and DS).[2] There are also three predicates (teacherOf (to), advisor (ad) and takeCourse (tc)) to link them. Two types of indexes, *predicate* and *type*, are added to accelerate query processing on RDF graph [51].

SPARQL, a W3C recommendation, is a standard query language developed for RDF graphs, which defines queries regarding graph patterns (GP). The principal part of SPARQL queries is as follows:

$$\mathsf{Q} := \mathsf{SELECT\ RD\ WHERE\ GP}$$

where (RD) is the *result description* and GP consists of *triple patterns* (TP). The triple pattern looks like a normal triple except that any *constant* can be replaced by a *variable* (e.g., ?X) to match a subgraph. The result description RD contains a subset of variables in the triple patterns (TP) to define the query results. For example, the query Q_H in Fig. 1 asks for professors (?X), courses (?Y) and students (?Z) such that the professor advises (ad) the student who also takes a course (tc) taught by (to) the professor. After exploring all three TPs in Q_H on the sample graph in Fig. 1, the exact match of RD (?X, ?Y and ?Z) is only a binding of Logan, OS, and Bobby.

Query processing on CPU. There are two representative approaches adopted in state-of-the-art RDF systems, (relational) triple join [40, 58, 12, 23] and graph exploration [62, 51]. A recent study [51] found that graph exploration with full-history pruning can provide low latency and high throughput for concurrent query processing. Therefore, we illustrate this approach to demonstrating the query processing on CPU with the sample RDF graph and SPARQL query (Q_H) in Fig. 1.

As shown in Fig. 2, all triple patterns of the query (Q_H) will be iterated in sequence (❶) to generate the results (*history table*) by exploring the graph, which is stored in an in-memory key/value store. According to the variable (?Y) of the current triple pattern (TP-2), each row of a certain column in the history table (❷) will be combined with the constant (takesCourse) of the triple pattern as

[2] Since the special predicate type (ty) is used to group a set of entities, we follow Wukong [51] to treat every type (e.g., professor (P)) as an index, the same as predicates (e.g., advisor (ad)).

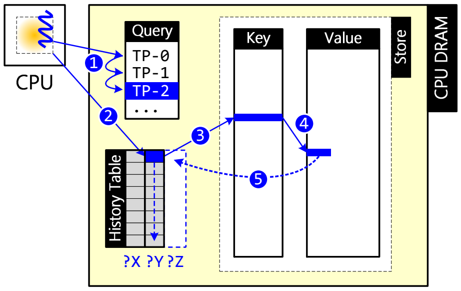


Fig. 2: The execution flow of query processing on CPU.

the key (❸) to retrieve the value (❹). The value will be appended to a new column (?Z) of the history table (❺). Note that an extra triple pattern (TP-0) from an index vertex (teacherOf) will be used to collect all start vertices satisfying a variable (?X) in TP-1.

Full-history pruning. Since processing RDF query by graph exploration needs to traverse the RDF graph, it is crucial to prune infeasible paths for better performance. There are basically two approaches: partial-history pruning [62], by inheriting partial history information (intermediate results) from previous steps of traversing to prune the following traversal paths; and full-history pruning [51], by passing the history information of all previous traversal steps for pruning. Wukong has exploited full-history pruning to prune unnecessary intermediate results precisely and *make all traversal paths completely independent*. Thanks to the fast RDMA-capable network as well as the relative cost-insensitivity of one-sided RDMA operations regarding payload size, full-history pruning is very effective and efficient to handle *concurrent* queries.

Workload heterogeneity. Prior work [62, 23, 51] has observed that there are two distinct types of SPARQL queries: *light* and *heavy*. Light queries (e.g., Q_L in Fig. 1) usually start from a (constant) normal vertex and only explore *a few* paths regardless of the dataset size. In contrast, heavy queries (e.g., Q_H in Fig. 1) usually start from an (type or predicate) index vertex and explore *massive amounts* of paths, which increases along with the growth of dataset size. The top of Fig. 3 demonstrates the number of paths explored by two typical queries (Q5 and Q7) on LUBM-10240 (10 vs. 16,000,000).

The *heterogeneity* in queries can result in tremendous latency differences on state-of-the-art RDF stores [51], even reaching more than 3,000X (0.13ms and 390ms for Q5 and Q7 on LUBM-10240 accordingly).[3] Therefore, the multi-threading mechanism is widely used by prior work [23, 62, 51] to improve the performance of heavy queries. However, such approach is intrinsically restricted by the limited computation resource of CPU. Currently, the maximum number of cores in a commercial CPU processor is usually less than 16. Moreover, the lengthy queries will significantly extend the latency of light queries and impair the throughput of processing concurrent queries. Some CPU systems like Oracle PGX [4] try to address this issue by adopting priority mechanism. However, with no variation of computing power, the sacrifice of user experience for one type of queries is unavoidable.

2.2 Hardware Trends

Hardware heterogeneity. With the prevalence of computational workloads (e.g., machine learning and data mining applications), it is now not uncommon to see server-class machines equipped with GPUs in the modern datacenter. As a landmark difference compared to CPU, the number of GPU cores (threads) can easily exceed two thousand, which far exceeds existing multicore CPU processors. As shown in Fig. 3, in a typical *heterogeneous* (CPU/GPU) machine, CPU and GPU have their private memory (DRAM) connected by PCIe with limited bandwidth (10GB/s). Compared to host memory (CPU DRAM), device memory (GPU DRAM) has much higher bandwidth (288GB/s vs. 68GB/s) but less capacity (12GB vs. 128GB). Generally, GPU is optimized for performing massive, simple and independent operations with intensive accesses on a relatively small memory footprint.

Fast communication: GPUDirect with RDMA. GPUDirect is a family of technologies that is continuously developed by NVIDIA [3]. Currently, it can support various efficient communications, including inter-node, intra-node, and inter-GPU. RDMA (Remote Direct Memory Access) is a networking feature to directly access the memory of a remote machine, which can bypass remote CPU and operating system, and avoid redundant memory copy. Hence, it has unique features like high speed, low latency and low CPU overhead. GPUDirect RDMA has been introduced in NVIDIA Kepler-class GPUs, like Tesla and Quadro series. This technique enables direct data transfer between GPUs by InfiniBand NICs as the name suggests [2].

2.3 Opportunities

Though prior work (e.g., Wukong [51]) has successfully demonstrated the low latency and high throughput of running light queries solely by leveraging graph exploration with full-history pruning, it is still incompetent to handle heavy queries efficiently. This leads to suboptimal performance when facing hybrid workloads comprising both light and heavy queries.

This problem is not due to the design and implementation of existing state-of-the-art systems, which have been heavily optimized by several approaches including multi-threading [62, 23, 51] and work-stealing scheme [51]. We attribute the performance issues mainly to the lim-

[3] Detailed experimental setup and results can be found in §6.

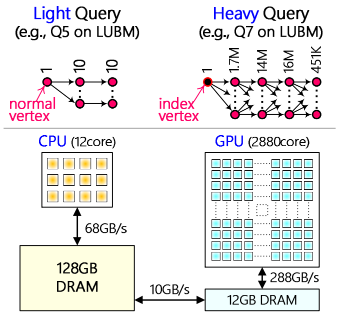


Fig. 3: Motivation of Wukong+G

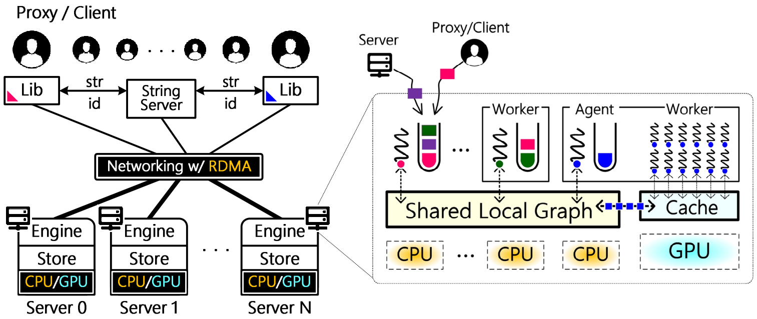


Fig. 4: The architecture overview of Wukong+G.

itation of handling hybrid workloads (light and heavy queries) on the homogeneous hardware (CPU), which can provide neither sufficient computation resources (a few cores) nor efficient data accesses (low bandwidth).

GPU is a good candidate to host heavy queries. First, the graph exploration strategy for query processing heavily relies on traversing massive paths on the graph store, which is a typical memory-intensive workload targeted by GPU's high memory bandwidth. Second, the memory latency hiding capability of GPU is inherently suitable for the random traversal on RDF graph, which is notoriously slow due to poor data locality. Third, every traversal path with the full-history pruning scheme is entirely independent, which can be fully parallelized on thousands of GPU cores.

In summary, the recent trend of hardware heterogeneity (CPU/GPU) opens an opportunity for running different queries on different hardware; namely, *running light queries on CPUs and heavy queries on GPUs*.

3 Wukong+G: An Overview

System architecture. An overview of Wukong+G's architecture is shown in Fig. 4. Wukong+G assumes running on a modern cluster connected with RDMA-capable fast networking, where each machine is equipped with one or more GPU cards. The GPU's device memory is treated as a cache for the large pool of the CPU's host memory. Wukong+G targets various SPARQL queries over a large volume of RDF data; it scales by partitioning the RDF graph into a large number of shards across multiple servers. Wukong+G may duplicate edges to make sure each server contains a self-contained subgraph (e.g., no dangling edges) of the input RDF graph for better locality. Note that there are no replicas of vertices in Wukong+G as no vertex data needs to synchronize. Moreover, Wukong+G also creates index vertices [51] for types and predicates to assist query processing.

Similar to prior work [51], Wukong+G follows a decentralized, shared-nothing, main-memory model on the server side. Each server consists of two separate layers: query engine and graph store. The query engine layer employs a worker-thread model by running N *worker threads* atop N CPU cores and dedicates one CPU core to run an *agent thread*; the agent thread will assist the worker threads on GPU cores to run queries. Each worker/agent thread on CPU has a task queue to continuously handle queries from clients or other servers, one at a time. The graph store layer adopts an RDMA-friendly key/value store over a distributed hash table to support a partitioned global address space. Each server stores a partition of the RDF graph, which is shared by all of worker/agent threads on the same server.

Wukong+G uses a set of dedicated proxies to run the client-side library and collect queries from massive clients. Each proxy parses queries into a set of stored procedures and generates optimal query plans using a cost-based approach. The proxy will further use the *cost* to classify a query into one of two types (light or heavy), and deliver it to a worker or agent thread accordingly.[4]

Basic query processing on GPU. In contrast to the query processing on CPU, which has to perform a triple pattern with massive paths in a verbose loop style (see Fig. 2), Wukong+G can fully parallelize the graph exploration with thousands of GPU cores. The basic approach is to dedicate one CPU core to perform the *control-flow* of the query, and use massive GPU cores to parallelize the *data-flow* of the query. As shown in Fig. 5, the agent thread on CPU core will first read the next triple pattern (❶) of the current query and prepare a cache of RDF datasets on GPU memory (❷). After that, the agent thread will leverage all GPU cores to perform the triple pattern in parallel (❸). Each worker thread on GPU core can independently fetch a row in the history table (❹) and combine it with the constant (takesCourse) of the triple pattern (TP-2) as the key (❺). The value retrieved by the key (❻) will be appended to a new column (?Z) of the history table (❼). While the hybrid design seems intuitive, Wukong+G still faces three challenges to run SPARQL queries on GPUs, which will be addressed by the techniques in §4:

[4] The recent release of Wukong (https://github.com/SJTU-IPADS/wukong) introduced a new cost-based query planner for graph exploration, where the cost estimation is roughly based on the number of paths may be explored. Wukong+G uses a user-defined threshold for the cost to classify queries.

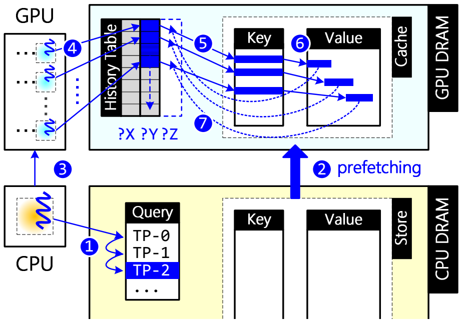


Fig. 5: The execution flow of query processing on CPU/GPU.

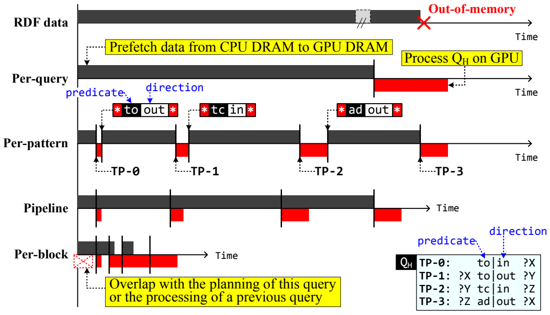


Fig. 6: The timeline of processing sample query (Q_H) on GPU.

C1: Small GPU memory. It is well known that GPU can obtain optimal performance only when the device memory (GPU DRAM) gets everything ready. Prior systems [62, 23, 51] can store an entire RDF graph in the host memory (CPU DRAM) since it is common that server-class machines equip with several hundred GBs of memory. However, this assumption does not apply to GPU since its current memory size usually stays less than 16GB. We should not allow device memory size to limit the upper bound of the supported working sets.

C2: Limited PCIe bandwidth. The memory footprint of SPARQL queries may touch arbitrary triples of the RDF graph. Therefore, the data transfer between CPU and GPU memory during query processing is unavoidable, especially for concurrent query processing. However, GPUs are connected to CPUs by PCIe (PCI Express), which has insufficient memory bandwidth (10GB/s). To avoid the bottleneck of data transfer, we should carefully design mechanisms to predict access patterns and minimize the number, volume and frequency of data swapping.

C3: Cross-GPU communication. With the increasing scale of RDF datasets and the growing number of concurrent queries, it is highly demanding that query processing systems can scale to multiple machines. Prior work [57, 51] has shown the effectiveness and efficiency of the partitioned RDF store and the worker-thread model. However, the intra-/inter-node communication between multiple GPUs has a long path: 1) device-to-host via PCIe; 2) host-to-host via networking; 3) host-to-device via PCIe. We should customize the communication flow for various participants to reduce the latency of network traffic.

4 Design

4.1 Efficient Query Processing on GPU

Facing the challenges like small GPU memory and limited PCIe bandwidth, we propose the following three key techniques to overcome them.

Query-aware prefetching. With the increase of RDF datasets, the limited GPU memory size (less than 16GB) is not enough to host the entire RDF graph. Wukong+G thus treats the GPU memory as a cache of CPU memory, and only ensures the necessary data is retained in GPU memory before running a query. However, it is non-trivial to decide the working set of a query accurately.

As shown in the second timeline of Fig. 6, Wukong+G proposes to just prefetch the triples with the *predicates* involved in a query, which can enormously reduce the memory footprint of a query from the entire RDF graph to the *per-query* scale. This assumption is based on two observations: 1) each query only touches a part of RDF graph; 2) the predicate of a triple pattern is commonly known (i.e., $\langle ?X, predicate, ?Y \rangle$). For example, the sample query (Q_H) only requires three predicates (teacherOf, takesCourse, and advisor), occupying about 3.7GB memory (0.3GB, 2.9GB, and 0.5GB respectively) for LUBM-2560.

Pattern-aware pipelining. For a query with many triple patterns, the total memory footprint of a single query may still exceed the GPU memory size. Fortunately, we further observe that the triple patterns of a query will be executed in sequence. It implies that Wukong+G can further reduce the demand for memory to the *per-pattern* scale. As shown in the third timeline of Fig. 6, Wukong+G can only prefetch the triples with a certain *predicate* that is used by the triple pattern will be immediately executed. Thus, for the sample query (Q_H) on LUBM-2560, the demand for GPU memory will further reduce to 2.9GB, the size of the maximum predicate (takesCourse).

Moreover, since the data prefetching and query processing are split into multiple independent phases, Wukong+G can use a *software pipeline* to create parallelism between the execution of the current triple pattern and the prefetching of the next predicate, as shown in the fourth timeline of Fig. 6. Note that it will also increase the memory footprint to the maximum size of two successive predicates (takesCourse and advisor).

Fine-grained swapping. Although the pattern-aware pipelining can overlap the latency of data prefetching and query processing, it is hard to perfectly hide the I/O cost due to limited bandwidth between system and

Table 1: A summary of optimizations for query processing on GPU. "$X|Y$" indicates XGB memory footprint and YGB data transfer. (†) The numbers are evaluated on 6GB GPU memory.

Granularity	Main Techniques	Q7 (GB) on LUBM-2560
Entire graph	Basic query processing	16.3 | 16.3
Per-query	Query-aware prefetching	5.6 | 5.6
Per-pattern	Pattern-aware pipelining	2.9 | 5.6
Per-block	Fine-grained swapping	2.9 | 0.7†

device memory (e.g., 10GB/s). For example, prefetching 2.9GB triples (takesCourse) requires about 300ms, which is even longer than the whole query latency (100ms). Therefore, Wukong+G adopts a fine-grained swapping scheme to maintain the triples cached in GPU memory. All triples with the same predicate will be further split into multiple fixed-size blocks, and the GPU memory will cache the triples in a best-effort way (§4.2). Consequently, the demand of memory will be further reduced to the *per-block* scale.

Moreover, the data transferring cost will also become the *per-block* scale, and all cached data on GPU memory can be reused by multiple triple patterns of the same query or even multiple queries. As shown in the fifth timeline of Fig. 6, when most triples of the required predicates have been retained in GPU memory, the prefetching cost can be perfectly hidden by query processing. Even for the first required predicate, Wukong+G still can hide the cost by overlapping it with the planning time of this query or the processing time of a previous query.

Table 1 summarizes the granularity of data prefetching on GPU memory, and shows the size of memory footprint and data transfer for a real case (Q7 on LUBM-2560). Note that Q7 is similar to Q_H but requires five predicates. The memory footprint of Q7 with fine-grained swapping is equal to the available GPU memory size (6GB) since Wukong+G only swaps out the triples of predicates on demand.

4.2 GPU-friendly RDF Store

Prior work [62, 51, 64] uses a distributed in-memory key/value store to physically store the RDF graph, which is efficient to support random traversals in graph-exploration scheme. In contrast to the intuitive design [62] that simply uses vertex ID (*vid*) as the key, and the in-/out-edge list (each element is a $[pid, vid]$ pair) as the value, Wukong [51] uses a combination of the vertex ID (*vid*), predicate ID (*pid*) and in/out direction (*d*) as the key (in the form of $[vid, pid, d]$), and the list of neighboring vertex IDs as the value (e.g., $[Logan, to, out] \longmapsto [DS]$ in the left part of Fig. 7).

This design can prominently reduce the graph traversal cost for both local and remote accesses. However, the triples (both key and value) with the same predicate are still sprinkled all over the store. It implies that the cost of prefetching keys and values for a triple pattern is extremely high or even impossible. Therefore, the *key/value store* on CPU memory should be carefully re-organized for heterogeneous CPU/GPU processing by aggregating all triples with the same predicate and direction into a *segment*. Furthermore, the key and value segments should be maintained in a fine-grained way (*block*) and be cached *in pairs*. Finally, the mapping between keys and values should be retained in the *key/value cache* on GPU memory, which uses a separate address space. Wukong+G proposes the following three new techniques to construct a GPU-friendly key/value store, as shown in the right part of Fig. 7.

Predicate-based grouping (CPU memory). Based on the idea of predicate-based decomposition in Wukong, Wukong+G adopts *predicate-based grouping* to exploit the predicate locality of triples and retains the encoding of keys and values. The basic idea is to partition the key space into multiple *segments*, which are identified by the combination of predicate and direction (i.e., $[pid, d]$). To preserve the support of fast graph exploration, Wukong+G still uses the hash function within the segment but changes the parameter from the entire key (i.e., $[vid, pid, d]$) to the vertex ID (*vid*). The number of keys and values in each segment are collected during loading the RDF graph and aligned to an integral multiple of the granularity of data swapping (*block*). To ensure that all values belonged to the triples with the same predicate are stored contiguously, Wukong+G groups such triples and inserts them together. Moreover, Wukong+G uses an indirect mapping to link keys and values, where the link is an *offset* within the value space instead of a direct pointer. As shown in the right part of Fig. 7, the triples required by TP-2 (i.e., $\langle Kurt, tc, DS\rangle$ and $\langle Bobby, tc, OS\rangle$) are aggregated together in both key and value spaces (the purple boxes).

Pairwise caching (GPU memory). To support fine-grained swapping, Wukong+G further splits each segment into multiple fixed-size blocks and stores them into *discontinuous* blocks of the cache on GPU memory, like $[Logan, to, out]$ and $[Erik, to, out]$. Note that the block size for keys and values can be different. Wukong+G follows the design on CPU memory to cache key and value blocks into separate regions on GPU memory, namely key cache and value cache. Wukong+G uses a simple table to map key and value blocks, and the link from key to value becomes the offset within the value block. Unlike the usual cache, the linked key and value blocks must be swapped in and out the (GPU) cache *in pairs*, like $[OS, tc, in]$ and $[Bobby]$ (the purple boxes). Thus, Wukong+G maintains a *bidirectional* mapping between the pair of cached key and value blocks. Moreover, a mapping table of block ID between RDF store (CPU) and cache (GPU) is used to re-translate the link between keys and values,

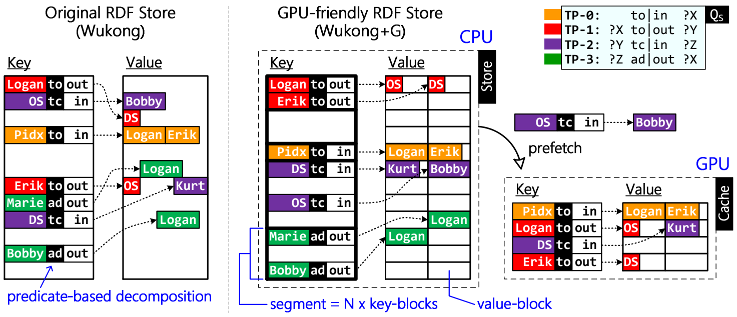


Fig. 7: The structure of GPU-friendly RDF store.

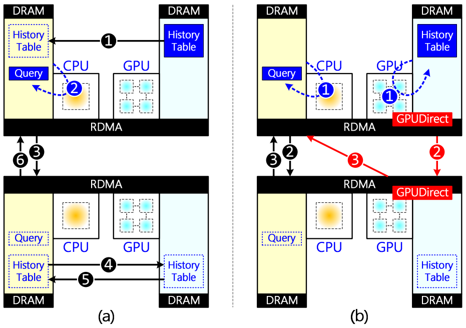


Fig. 8: Communication flow w/o and w/ GPUDirect.

when the pairwise blocks (key and value) are swapped in GPU memory.

Look-ahead replacement policy. The mapping table of block IDs between RDF store and cache records whether the block has been cached. Before running a triple pattern of the query, all of key and value blocks should be prefetched to the GPU memory. For example, the key $[OS, tc, in]$ and value $[Bobby]$ should be loaded into the cache before processing TP-2. Wukong+G proposes a *look-ahead LRU-based replacement policy* to decide where to store prefetched key and value blocks. Specifically, Wukong+G prefers to use free blocks first and then chooses the blocks that will not be used by the following triple patterns of this query (*look-ahead*), with the highest LRU score. The worst choice is the blocks will be used by the following triple patterns, and then the block of the farthest triple pattern will be replaced. Note that the replacement policies for keys and values are the same and there is at most a pair of key/value blocks will be swapped out due to the pairwise caching scheme.

For example, as shown in the right part of Fig. 7, before running the triple pattern TP-2, all key/value blocks of the predicate takeCourse (tc) should be swapped in the cache (the purple boxes). The value block with $[Bobby]$ can be loaded to a free block, while the key block with $[OS, tc, in]$ will replace the cached block with $[Pidx, to, in]$, since it was used by TP-0 with the highest LRU score.

4.3 Distributed Query Processing

Wukong+G splits the RDF graph into multiple disjoint partitions by a differentiated partitioning algorithm [51, 19] [5], and each machine hosts an RDF graph partition and launches many worker threads on CPUs and GPUs to handle concurrent light and heavy queries respectively. The CPU worker threads on different machines will only communicate with each other for (light) query processing, and it is the same to GPU worker threads for (heavy) query processing.

To handle light queries on CPU worker threads, Wukong+G simply follows the procedure (see Fig. 2) that has been successfully demonstrated by Wukong [51]. However, to handle heavy queries on GPU worker threads, the procedure (see Fig. 5) becomes complicated due to the assistance of (CPU) agent thread and the maintenance of (GPU) RDF cache.

Execution mode: fork-join. Prior work [51] proposes two execution modes, *in-place* and *fork-join*, for distributed graph exploration to migrate data and execution respectively. The in-place execution mode synchronously leverages one-sided RDMA `READ` to directly fetch data from remote machines, while the fork-join mode asynchronously splits the following query computation into multiple sub-queries running on remote machines. Wukong+G follows the design on CPU worker threads but only adopts the fork-join mode for query processing on GPU, because the in-place mode is usually inefficient for heavy queries [51] and migrating data from remote CPU memory to local GPU memory is still very costly even with RDMA operations.

In the fork-join mode, the agent thread will split the running query (metadata) with intermediate results (history table) into multiple sub-queries for the following query processing, and dispatch them to the task queue of agent threads on remote machines by leveraging one-sided RDMA `WRITE`. Therefore, multiple heavy queries can be executed on multiple GPUs concurrently in a time-sharing way. However, the current history table is located in GPU memory (see Fig. 8), such that it would be inefficient to fetch and split the table by using a single agent thread on CPU (❶ and ❷ in Fig. 8(a)). Therefore, Wukong+G leverages all GPU cores to partition the history table in fully parallel (❶ in Fig. 8(b)) using a dynamic task scheduling mechanism [47, 18].

Communication flow. To support fork-join execution, the sub-queries will be sent to target machines with their metadata (e.g., query plan and current step) and history table (intermediate results), and the history table will be sent back with final results at the end. As shown in Fig. 8(a), the query metadata will be delivered by one-sided RDMA operations between the CPU memory of

[5] The normal vertex (e.g., Logan) will be assigned to only one machine with all of its edges, while the index vertex (e.g., teacherOf) will be split and replicated to multiple machines with edges linked to normal vertices on the same machine.

Table 2: A collection of synthetic and real-life datasets. #T, #S, #O and #P mean the number of triples, subjects, objects and predicates respectively. (†) The size of datasets in raw NT format.

Dataset	#T	#S	#O	#P	Size†
LUBM-2560	352 M	55 M	41 M	17	58GB
LUBM-10240	1,410 M	222 M	165 M	17	230GB
DBPSB	15 M	0.3 M	5.2 M	14,128	2.8GB
YAGO2	190 M	10.5 M	54.0 M	99	13GB

two machines (❸ and ❻). In contrast, the history table has to go through a long path from local GPU memory to the remote GPU memory, and finally goes back to the local CPU memory. A detailed communication flow for history table (see Fig. 8(a)): 1) from local GPU memory to local CPU memory (❶, Device-to-Host); 2) from local CPU memory to remote CPU memory (❸, Host-to-Host); 3) from remote CPU memory to remote GPU memory (❹, Host-to-Device); 4) from remote GPU memory to remote CPU memory (❺, Device-to-Host); 5) from local CPU memory to remote CPU memory (❻, Host-to-Host).

GPUDirect [3] opens an opportunity for Wukong+G to directly write history table from local GPU memory to remote GPU and CPU memory. Hence, Wukong+G decouples the transferring process of query metadata and history table (❷ and ❷ in Fig. 8(b)), and further shortens the communication flow for history table by leveraging GPUDirect RDMA. It also avoids the contention on agent thread with the metadata transferring. A detailed communication flow for history table (see Fig. 8(b)): 1) from local GPU memory to remote GPU memory (❷, Device-to-Device); 2) from remote GPU memory to local CPU memory (❸, Device-to-Host).

Moreover, to mitigate the pressure on GPU memory when handling multiple heavy queries, Wukong+G choose to send the history table of pending queries from local GPU memory to the buffer on remote CPU memory first via GPUDirect RDMA, and delay the prefetching of history table from CPU memory to GPU memory till handling the query on GPU.

5 Implementation

Wukong+G prototype is implemented in 4,088 lines of C++/CUDA codes atop of the code base of Wukong. This section describes some implementation details.

Multi-GPUs support. Currently, it is not uncommon to equip every CPU socket with a separate GPU card for low communication cost and good locality. To support such multi-GPUs on a single machine, Wukong+G runs a separate server for each GPU card and several co-located CPU cores (usually a socket). All servers comply with the same communication mechanism via GPUDirect-capable RDMA operations, regardless of whether two servers share the same physical machine or not.

Too large intermediate results. In rare cases, the intermediate results may overflow the history buffer on GPU memory. For example, we witness this scenario in YAGO2 benchmark (§6.8) that a heavy query keeps spanning out without any pruning. Wukong+G can horizontally divide the intermediate results into multiple strips by row and only hold a single strip into the history table on GPU memory. The remaining strips will stay in CPU memory and be swapped in GPU memory one-by-one while processing a single triple pattern.

Table 3: The query performance (msec) on a single server.

LUBM-2560		TriAD	Wukong	Wukong+G
H	**Q1** (3.6GB)	851	992	**165**
	Q2 (2.4GB)	211	138	**31**
	Q3 (3.6GB)	424	340	**63**
	Q7 (5.6GB)	2,194	828	**100**
	Geo. M	639	443	**75**
L	**Q4**	1.45	**0.13**	0.16
	Q5	1.10	**0.09**	0.11
	Q6	16.67	**0.49**	0.51
	Geo. M	2.98	**0.18**	0.21

6 Evaluation

6.1 Experimental Setup

Hardware configuration. All evaluations are conducted on a rack-scale cluster with 10 servers on 5 machines. We run two servers on a single machine. Each server has one 12-core Intel Xeon E5-2650 v4 CPU with 128GB of DRAM, one NVIDIA Tesla K40m GPU with 12GB of DRAM, and one Mellanox ConnectX-3 56Gbps InfiniBand NIC via PCIe 3.0 x8 connected to a Mellanox IS5025 40Gbps IB Switch. Wukong+G only provides a one-to-one mapping between the work and agent threads on different servers [51], which mitigates the scalability issue of RDMA networks with reliable transports [31] and simplifies the implementation of the task queue. In all experiments, we reserve two cores on each CPU to generate requests for all servers to avoid the impact of networking between clients and servers as done in prior work [54, 56, 57, 20, 51].

Benchmarks. Our benchmarks include one synthetic and two real-life datasets, as shown in Table 2. The synthetic dataset is the Lehigh University Benchmark (LUBM) [7]. We generate 5 datasets with different sizes (up to LUBM-10240) and use the query set published in Atre et al. [13], which are widely used by many distributed RDF systems [36, 62, 23, 51]. The real-life datasets include the DBpedia's SPARQL Benchmark (DBPSB) [1] and YAGO2 [9, 30]. For DBPSB, we use the query set recommended by its official site. For YAGO2, we collect our query set from both H_2RDF+ [43] and RDF-3X [42] to make sure the test covers both light and heavy queries.

Comparing targets. We compare our system against two state-of-the-art distributed RDF query systems, TriAD [23] (RDF relational store) and Wukong [51] (RDF graph stores). Note that TriAD does not sup-

Table 4: The query performance (msec) on 10 servers.

LUBM-10240		TriAD	Wukong	Wukong+G
H	**Q1** (14.25GB)	3,400	480	**211**
	Q2 (9.74GB)	880	66	**12**
	Q3 (14.25GB)	2,835	171	**19**
	Q7 (22.58GB)	10,806	390	**100**
	Geo. M	3,094	215	**47**
L	**Q4**	3.08	**0.44**	0.46
	Q5	1.84	**0.13**	0.17
	Q6	65.20	**0.70**	0.71
	Geo. M	7.04	**0.34**	0.38

port concurrent query processing, so we only compare to it in the single query performance. As done in prior work [62, 23, 51], the string server is enabled for all systems to save memory usage, reduce network bandwidth, and boost string matching.

6.2 Single Query Performance

We first study the performance of Wukong+G for a single query using the LUBM dataset. Table 3 shows the optimal performance of different systems on a single server with LUBM-2560. For Wukong+G, there is no data swapping during single query experiment since the current memory footprint of all queries on LUBM-2560 (the numbers in brackets) is smaller than the GPU memory (12GB). The query-aware prefetching reduces the memory footprint to the per-query granularity (see Table 1).

Although Wukong and TriAD have enabled multi-threading (10 worker threads), Wukong+G can still significantly outperform such pure CPU systems for heavy queries (Q1-Q3, Q7) by up to 8.3X and 21.9X (from 4.5X and 5.2X) due to wisely leveraging hardware advantages. The improvement of average (geometric mean) latency reaches 5.9X and 8.5X. For the light queries (Q4-Q6), Wukong+G inherits the prominent performance of Wukong by leveraging graph exploration and outperforms TriAD by up to 32.7X.

We further compare Wukong+G with Wukong and TriAD (multi-threading enabled) on 10 servers using LUBM-10240 in Table 4. For heavy queries, Wukong+G still outperforms the average (geometric mean) latency of Wukong by 4.6X (ranging from 2.3X to 9.0X), thanks to the heterogeneous RDMA communication for preserving the good performance of GPU at scale. Further, using up all CPU worker threads to accelerate a single query is not practical for concurrent query processing since it will result in throughput collapse. For light queries, Wukong+G incurs about 12% performance overhead (geometric mean) compared to Wukong due to adjusting the layout of key/value store on CPU memory for predicate-based grouping. Wukong+G is still one order of magnitude faster than TriAD due to the in-place execution with one-sided RDMA `READ` [51].

Table 5: The contribution of (cumulative) optimizations to the query latency (msec) evaluated on 3GB GPU memory.

LUBM-2560	Per-query	Per-parttern	Per-block	Pipeline
Q1 (3.6GB)	x	743	313	295
Q2 (2.4GB)	284	283	32	31
Q3 (3.6GB)	x	309	62	63
Q7 (5.6GB)	x	893	622	610

6.3 Factor Analysis of Improvement

To study the impact of each technique and how they affect the query performance, we iteratively enable each optimization and collect the average latency by repeatedly running the same query on a single server with 3GB GPU memory for LUBM-2560. As shown in Table 5, even using query-aware prefetching (per-query), the memory footprints of query Q1, Q3 and Q7 still exceed available GPU memory (see Table 3). Hence, they can not run until enabling pattern-aware prefetching (per-pattern). The effectiveness of fine-grained swapping (per-block) varies on different queries. It is quite effective on Q2 and Q3 (8.8X and 5.0X) since all triples required by triple patterns can almost be stored in 3GB GPU memory. Note that Q3 returns an empty history table (intermediate results) half-way and reduces the practical runtime memory footprint to 2.5GB. For Q1 and Q7, although the relative large memory footprint (3.6GB and 5.6GB), incurs massive data swapping (1.5GB by 187 time and 5.1GB by 734 times), the cache sharing with fine-grained mechanism can still notably reduce the query latency by 2.4X and 1.4X. Moreover, pipeline does not work on Q2 and Q3 without data prefetching time. The improvement for Q1 and Q7 is still limited since the prefetching and execution time for each triple pattern are quite imbalanced. For example, 88% of blocks are swapped at two triple patterns for Q7.

Table 6: A comparison of query performance (msec) w/o and w/ GPUDirect RDMA (GDR) on 10 servers with LUBM-10240.

LUBM-10240	Q1	Q2	Q3	Q7
Wukong+G w/o GDR	222 (53.4)	13	22	103 (26.1)
Wukong+G w/ GDR	211 (40.1)	13	22	98 (21.3)

6.4 GPUDirect RDMA

To shorten communication flow and avoid redundant memory copy for history table (intermediate results) of queries, Wukong+G leverages GPUDirect RDMA (GDR) to write history table directly from local GPU memory to remote GPU and CPU memory (§4.3). To study the impact of leveraging GPUDirect RDMA, we enforce Wukong+G to purely use native RDMA for both query metadata and history table (i.e., Wukong+G w/o GDR). As shown in Table 6, the performance of Q2 and Q3 is non-sensitive to GPUDirect RDMA because of no data transfer among GPUs. For Q1, leveraging GPUDirect RDMA can reduce about 30% communication cost (53.4ms vs. 40.1ms), since the query need to send about 487MB intermediate results by about 990 times RDMA operations. For queries with relatively large intermediate results or many triple patterns, there are more rooms for the overall performance improvement.

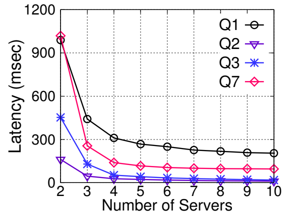


Fig. 9: The latency with the increase of servers.

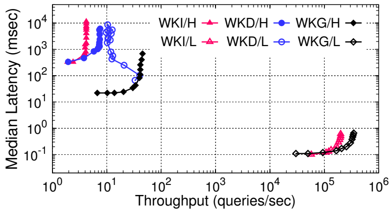


Fig. 10: The performance of hybrid workload on 10 servers with LUBM-10240.

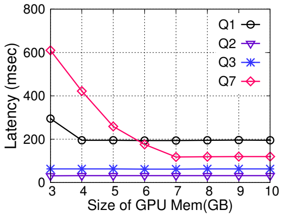


Fig. 11: The latency with the increase of GPU memory.

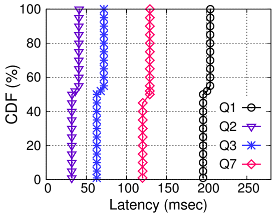


Fig. 12: The CDF of latency for mixed heavy workload.

6.5 Scalability

We evaluate the scalability of Wukong+G with the increase of servers. Since the latency of light queries of Wukong+G mainly inherits from Wukong, We only report the experimental results of heavy queries handled by GPUs. As shown in Fig. 9, the speedup of heavy queries ranges from 4.8X to 23.8X. As the number of servers increases from 2 to 10, a good horizontal scalability is shown. After a detailed analysis of the experimental results, we reveal that there are two different factors improving the performance at different stages. In the first stage (from 2 to 4 servers), the increase of total GPU memory provides the main contribution to the performance gains, ranging from 3.2X to 8.5X, by reducing memory swapping cost. In the second stage (from 4 to 10 servers), since Wukong+G stops launching expensive *memory swapping* operations when enough GPU memory is available, the main performance benefits come from using more GPUs, ranging from 1.5X to 2.8X.

Discussion. With the further increase of servers, the single query latency may not further decrease due to fewer workload per server and more communication cost. It implies that it is not worth making all resources (GPUs) participate in a single query processing, especially for a large-scale cluster (e.g., 100 servers). Therefore, Wukong+G will limit the participants of a single query and can still scale well in term of throughput by handling more concurrent queries simultaneously on different servers.

6.6 Performance of Hybrid Workloads

One principal aim of Wukong+G is to handle concurrent hybrid (light and heavy) queries in an efficient and scalable way. Prior work [51] briefly studied the performance of Wukong with a mixed workload, which consists of 6 classes of light queries (Q4-Q6 and A1-A3[6]). The light query in each class has a similar behavior except that the start point is randomly selected from the same type of vertices (e.g., Univ0, Univ1, etc.). The distribution of query classes follows the reciprocal of their average latency. Therefore, we first extend original mixed workload by adding 4 classes of heavy queries (Q1-Q3, Q7), and then allow all clients to freely send light and heavy queries[7] according to the distribution of query classes.

We compare Wukong+G (WKG) with two different settings of Wukong: Default (WKD) and Isolation (WKI). Wukong/Default (WKD) allows all worker threads to handle hybrid queries, while Wukong/Isolation (WKI) reserves half of the worker threads to handle heavy queries. Each server runs two emulated clients on dedicated cores to send requests. Wukong launches 10 worker threads, while Wukong+G launches 9 worker threads and an agent thread. The multi-threading for heavy queries is configured to 5. We run the hybrid workload over LUBM-10240 on 10 servers for 300 seconds (10s warmup) and report the throughput and median (50^{th} percentile) latency for light and heavy queries separately over that period in Fig. 10.

For heavy queries, Wukong+G improves throughput and latency by over one order of magnitude compared to Wukong (WKD and WKI). The throughput of WKD is notably better (about 80%) than that of WKI, since it can use all worker threads to handle heavy queries. For light queries, Wukong+G performs up to 345K queries per second with median latency of 0.6ms by 9 worker threads. The latency can be halved with a small 10% impact in throughput. As expected, WKI can provide about half of the throughput (199K queries/s) with a similar latency since only half of the worker threads (5) are used to handle light queries. However, the throughput and latency of WKD for light queries are thoroughly impacted by the processing of heavy queries.

6.7 RDF Cache on GPU

To study the influence of GPU cache for the performance of heavy queries on Wukong+G, we first evaluate the single query latency using LUBM-2560 on a single server with the GPU memory sizes varying from 3GB to 10GB. We repeatedly send one kind of heavy queries until the cache on GPU memory is warmed up, and illustrate the average latency of heavy queries in Fig. 11. Since the memory footprint of Q2 (2.4GB) is always smaller than the GPU memory, the latency is stable, and there is no data swapping. For Q3, although the memory footprint

[6] Three additional queries (A1, A2, and A3) are from the official LUBM website (#1, #3, and #5).

[7] In prior experiment [51], only up to one client is used to continually send heavy queries (i.e., Q1).

Table 7: The latency (msec) of queries on DBPSB and YAGO2

DBPSB	D1	D2	D3	D4	D5	Geo. M
Wukong	1.28	0.15	0.25	4.25	1.08	0.74
Wukong+G	0.53	0.16	0.26	0.99	0.52	0.41

YAGO2	Y1	Y2	Y3	Y4	Geo. M
Wukong	0.10	0.13	4685	752	14.6
Wukong+G	0.11	0.15	1856	398	10.5

of the query is about 3.6GB, the latency is still stable since the history table becomes empty after the first two triple patterns due to contradictory conditions, where the rest predicate segment (about 1.1GB) will never be loaded. For Q1 and Q7, the latency decreases with the increase of GPU memory due to the decrease of data swapping size. However, the break point of Q7 is later than that of Q1 since it has a relatively larger memory footprint (5.6GB vs. 3.6GB).

To show the effectiveness of sharing GPU cache by multiple heavy queries, We further evaluate the performance of a mixture of four heavy queries using LUBM-2560 on a single server with 10GB GPU memory. As shown in Fig. 12, the geometric mean of 50^{th} (median) and 99^{th} percentile latency is just 84.5 and 93.8 milliseconds respectively, under the peak throughput. Compared to the single query latency (see Table 3), the performance degradation is just 3% and 14%, thanks to our fine-grained swapping and look-ahead replacing. During the experiment, the number and volume of blocks swapped in per second are about 96 and 750MB.

6.8 Other Workloads

We further compare the performance of Wukong+G with Wukong on two real-life datasets, DBPSB [1] and YAGO2 [9]. As shown in Table 7, for light queries (D2, D3, Y1, and Y2), Wukong+G can provide a close performance to Wukong due to following the same execution mode and a similar in-memory store. For heavy queries (D1, D4, D5, Y3, and Y4), Wukong+G can notably outperform Wukong by up to 4.3X (from 1.9X).

7 Related Work

Wukong+G is inspired by and departs from prior RDF query processing systems [40, 58, 41, 12, 50, 13, 65, 60, 14, 61, 62, 23, 51, 32, 64], but differs from them in exploiting a distributed heterogeneous CPU/GPU cluster to accelerate heterogeneous RDF queries.

Several prior systems [11, 12] have leveraged column-oriented databases [53] and vertical partitioning for RDF dataset, which group all triples with the same predicate into a single two-column table. The predicate-based grouping in Wukong+G is driven by a similar observation. However, Wukong+G still randomly (hash-based) assign keys within the segment to preserve fast RDMA-based graph exploration, which plays a vital role for running light queries efficiently on CPU.

Using prefetching and pipelining mechanisms are not new, which have been exploited in many graph-parallel systems [49, 35] and GPU-accelerated systems [37] to hide the latency of memory accesses. Wukong+G employs a SPARQL-specific prefetching scheme and enables such techniques on multiple concurrent jobs (heavy queries) that share a single cache on the GPU memory.

There has been a lot of work [25, 24, 39, 26, 29, 27, 55, 46, 28] focusing on exploiting the unique features of GPUs to accelerate database operations. Mega-KV [63] is an in-memory key/value store that uses GPUs to accelerate index operations by only saving indexes on the GPU memory to ease device memory pressure. CoGaDB [15, 16] uses a column-oriented caching mechanism on GPU memory to accelerate OLAP workload. SABER [34] is a hybrid high-performance relational stream processing engine for CPUs and GPUs. Wukong+G is inspired by prior work, while the differences in workloads result in different design choices. To our knowledge, none of the above systems exploit distributed heterogeneous (CPU/GPU) environment, let alone using RDMA as well as GPUDirect features.

To reduce communication overhead between multiple GPUs, NVIDIA continuously puts forward GPUDirect technology [3], including GPUDirect RDMA and GPUDirect Async (under development [6]). They enable direct cross-device data transfer on data plane and control plane, respectively. Researchers have also investigated how to provide network [33, 21] and file system abstractions [52] based on such hardware features. Our design currently focuses on using GPUs to deal with heavy queries for RDF graphs. The above efforts provide opportunities to build a more flexible and efficient RDF query system through better abstractions.

8 Conclusions

The trend of hardware heterogeneity (CPU/GPU) opens new opportunities to rethink the design of query processing systems facing hybrid workloads. This paper describes Wukong+G, a graph-based distributed RDF query system that supports heterogeneous CPU/GPU processing for hybrid workloads with both light and heavy queries. We have shown that Wukong+G achieves low query latency and high overall throughput in the single query performance and hybrid workloads.

Acknowledgments

We sincerely thank our shepherd Howie Huang and the anonymous reviewers for their insightful suggestions. This work is supported in part by the National Key Research & Development Program (No. 2016YFB1000500), the National Natural Science Foundation of China (No. 61772335, 61572314, 61525204), the National Youth Top-notch Talent Support Program of China, and Singapore NRF (CREATE E2S2).

References

[1] DBpedias SPARQL Benchmark. http://aksw.org/Projects/DBPSB.

[2] Developing a Linux Kernel Module using GPUDirect RDMA. http://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[3] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[4] Parallel Graph AnalytiX (PGX). http://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytix/overview/index.html.

[5] Semantic Web. https://www.w3.org/standards/semanticweb/.

[6] State of GPUDirect Technologies. http://on-demand.gputechconf.com/gtc/2016/presentation/s6264-davide-rossetti-GPUDirect.pdf.

[7] SWAT Projects - the Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/.

[8] Wikidata. https://www.wikidata.org.

[9] YAGO: A High-Quality Knowledge Base. http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago.

[10] Bio2RDF: Linked Data for the Life Science. http://bio2rdf.org/, 2014.

[11] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB'07, pages 411–422, 2007.

[12] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB Journal*, 18(2):385–406, 2009.

[13] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW'10, pages 41–50, 2010.

[14] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD'13, pages 121–132, 2013.

[15] S. Breß. The design and implementation of CoGaDB: A column-oriented gpu-accelerated dbms. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[16] S. Breß, N. Siegmund, M. Heimel, M. Saecker, T. Lauer, L. Bellatreche, and G. Saake. Load-aware inter-co-processor parallelism in database query processing. *Data & Knowledge Engineering*, 93:60–79, 2014.

[17] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, et al. Tao: Facebooks distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC'13, pages 49–60, 2013.

[18] R. Chen, H. Chen, and B. Zang. Tiled-mapreduce: Optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT'10, pages 523–534, 2010.

[19] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys'15, pages 1:1–1:15, 2015.

[20] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys'16, pages 26:1–26:17, New York, NY, USA, 2016. ACM.

[21] F. Daoud, A. Watad, and M. Silberstein. GPUrdma: Gpu-side library for high performance networking from gpu kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS'16, pages 6:1–6:8, 2016.

[22] Google Inc. Introducing the knowledge graph: things, not strings. https://googleblog.blogspot.co.uk/2012/05/introducing-knowledge-graph-things-not.html, 2012.

[23] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: A distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD'14, pages 289–300, 2014.

[24] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[25] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD'08, pages 511–524, 2008.

[26] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proceedings of the VLDB Endowment*, 6(10):889–900, Aug. 2013.

[27] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled cpu-gpu architectures. *Proceedings of the VLDB Endowment*, 8(4):329–340, Dec. 2014.

[28] M. Heimel, M. Kiefer, and V. Markl. Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD'15, pages 1477–1492, 2015.

[29] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013.

[30] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum. Yago2: Exploring and querying world knowledge in time, space, context, and many languages. In *Proceedings of the 20th International Conference Companion on World Wide Web*, WWW'11, pages 229–232, 2011.

[31] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'16, 2016.

[32] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica. Zipg: A memory-efficient graph store for interactive queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD'17, pages 1149–1164, 2017.

[33] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. GPUnet: Networking abstractions for gpu programs. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, volume 14 of *OSDI'14*, pages 6–8, 2014.

[34] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD'16, pages 555–569, 2016.

[35] P. Kumar and H. H. Huang. G-store: high-performance graph store for trillion-edge processing. In *International Conference forHigh Performance Computing, Networking, Storage and Analysis*, SC'16, pages 830–841. IEEE, 2016.

[36] K. Lee and L. Liu. Scaling queries over big rdf graphs with semantic hash partitioning. *Proceedings of the VLDB Endowment*, 6(14):1894–1905, Sept. 2013.

[37] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai. Garaph: efficient gpu-accelerated graph processing on a single machine with balanced replication. In *2017 USENIX Annual Technical Conference*, USENIX ATC'17, pages 195–207. USENIX Association, 2017.

[38] National Center for Biotechnology Information. PubChemRDF. https://pubchem.ncbi.nlm.nih.gov/rdf/, 2014.

[39] S. I. F. G. N. Nes and S. M. S. M. M. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *Data Engineering*, 40, 2012.

[40] T. Neumann and G. Weikum. RDF-3X: A risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, Aug. 2008.

[41] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD'09, pages 627–640, 2009.

[42] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, Feb. 2010.

[43] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2rdf+: High-performance distributed joins over large-scale rdf graphs. In *2013 IEEE International Conference on Big Data*, IEEE BigData'13, pages 255–263, 2013.

[44] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2rdf: Adaptive query processing on rdf data in the cloud. In *Proceedings of the 21st International Conference on World Wide Web*, WWW'12 Companion, pages 397–400, 2012.

[45] Philip Howard. Blazegraph GPU. https://www.blazegraph.com/whitepapers/Blazegraph-gpu_InDetail_BloorResearch.pdf, 2015.

[46] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *Proceedings of the 2014 IEEE 30th International Conference on Data Engineering*, ICDE'14, pages 508–519, 2014.

[47] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA'07, pages 13–24, 2007.

[48] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, PSI EtA'10, pages 4:1–4:5, 2010.

[49] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 472–488, 2013.

[50] S. Sakr and G. Al-Naymat. Relational processing of rdf queries: A survey. *SIGMOD Record*, 38(4):23–28, June 2010.

[51] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 317–332, 2016.

[52] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: Integrating a file system with gpus. *ACM Trans. Comput. Syst.*, 32(1):1:1–1:31, Feb. 2014.

[53] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[54] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 18–32. ACM, 2013.

[55] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. *Proceedings of the VLDB Endowment*, 7(11):1011–1022, 2014.

[56] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, pages 26:1–26:15, New York, NY, USA, 2014. ACM.

[57] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[58] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, Aug. 2008.

[59] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: A probabilistic taxonomy for text understanding. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD'12, pages 481–492, 2012.

[60] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD'12, pages 517–528, New York, NY, USA, 2012. ACM.

[61] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: A fast and compact system for large scale rdf data. *Proceedings of the VLDB Endowment*, 6(7):517–528, May 2013.

[62] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 265–276, 2013.

[63] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015.

[64] Y. Zhang, R. Chen, and H. Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 614–630, 2017.

[65] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering sparql queries via subgraph matching. *Proceedings of the VLDB Endowment*, 4(8):482–493, May 2011.

MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through

Bo Peng
Shanghai Jiao Tong University
Intel Corporation

Haozhong Zhang
Intel Corporation

Jianguo Yao
Shanghai Jiao Tong University

Yaozu Dong
Intel Corporation

Yu Xu
Shanghai Jiao Tong University

Haibing Guan
Shanghai Jiao Tong University

Abstract

The fast access to data and high parallel processing in high-performance computing instigates an urgent demand on the I/O improvement of the NVMe storage within datacenters. However, unsatisfactory performance of the former NVMe virtualization demonstrates that NVMe storage devices are often underutilized within cloud computing platforms. NVMe virtualization with high performance and device sharing has captured the attention of researchers. This paper introduces MDev-NVMe, a new virtualization implementation for NVMe storage device with: (1) full NVMe storage virtualization running native NVMe driver in guest, and (2) a mediated pass-through mechanism with an active polling mode which can achieve both high throughput, low latency performance and a good device scalability. This paper subsequently evaluates MDev-NVMe on Intel OPTANE and P3600 NVMe SSD by comparison with the mainstream virtualization mechanisms using application-level I/O benchmarks. With polling, MDev-NVMe can demonstrate a 142% improvement over native (interrupt-driven) throughput and over 2.5 × the *Virtio* throughput with only 70% native average latency and 31% *Virtio* average latency with a reliable scalability. Finally, the advantages of MDev-NVMe and the importance of polling are discussed, offering evidence that MDev-NVMe is a superior virtualization choice with high performance and promising levels of maintenance.

1 Introduction

NVM Express (NVMe) [21] is an inherently parallel, high-performing interface and command set designed for non-volatile memory based storage. NVMe devices have been designed from a logical device interface where storage media is attached via a PCIe bus [26]. NVMe SSDs can deliver I/O operations with a very high throughput performance on the low latency, making them the superior storage choices for data-intensive datacenters to obtain high performance computing in cloud services [2, 5]. Currently, NVMe SSDs are also used to accelerate I/O missions between system storage and other PCIe devices, such as GPUs on graphics cards [7]. However, because of imbalanced requirements of applications in high performance platforms, NVMe devices are often underutilized in terms of I/O performance [19, 22, 17]. NVMe devices in datacenters require solutions to achieve high performance and sharing capabilities [31].

I/O virtualization [13, 24] is a key component of cloud computing, which not only optimizes utilization of physical resources, but also simplifies storage management, providing a simple and consistent interface to complex functions. Typical cases of NVMe virtualization include VM boot disk and server side caching of VM data. With respect to NVMe devices, limited PCIe slots make NVMe virtualization essential in datacenters with high density so that the benefits of NVMe devices can be shared among the VMs.

Recently, NVMe virtualization [33, 28, 3] has become widely studied by computer scientists and researchers. Keeriyadath [16] summarizes three of the NVMe virtualization approaches; implementing SCSI to NVMe translation layer on the hypervisor (blind mode), pure virtual NVMe stack by distributing I/O queues amongst hosted VMs (Virtual Mode), and SR-IOV [11] based NVMe controllers per virtual functions (physical mode). Modern virtualization solutions for NVMe Storage such as *Virtio* [23], Userspace NVMe driver in QEMU [34] and Storage Performance Development Kit (SPDK) [15] are implemented in the userspace of the Linux system. However, these mechanisms have their respective shortcomings: the performance of VMs with *Virtio* and Userspace NVMe driver in QEMU are considered poor compared with native drivers. At the same time SPDK must collaborate with “Hugepage” memory which adds further pressure to the memory resources of the host server.

Observing that mediated pass-through [30] has been gradually recognized as an effective solution for I/O virtualization [12, 27], concurrently the Linux kernel has introduced a mediated device framework which supports such usage since 4.10 [25]. Therefore, it is proposed that implementing an effective NVMe virtualization using mediated pass-through is an appropriate methodology. The detail of MDev-NVMe design is that a full virtualization is proposed which enables VMs running with a native driver, and a MDev-NVMe module with device emulation, admin queue emulation, and also with I/O queue shadow. To improve the VM performance, an active polling mode for queue handling is proposed. As a result, this design is then built for the experiments on Fio [4] benchmarks by undertaking comparison experiments between mainstream NVMe virtualization solutions. The proposed design can achieve up to 142% improvement of througerhput with 70% average latency over native performance, and also performs well regarding scalability and flexibility. Also, the experiments are performed on different I/O blocksizes and provide suggestions on selecting appropriate blocksizes for NVMe virtualization. At last, the discussion of optimization on active polling is undertaken in the final section, which also provides suggestions on the future design of virtualization and high-performance storage.

In summary, this paper makes the following contributions:

(1) We introduce a full NVMe virtualization solution MDev-NVMe with mediated pass-through that runs the native NVMe driver in each guest.

(2) We demonstrate details of emulation of Admin queues and shadow of I/O queues in the mediated pass-through which can pass-through performance-critical resources accesses in NVMe storage.

(3) We design an active polling for shadow SQ and CQs and host CQs in NVMe to achieve better performance over native devices.

(4) We do further comparison evaluations of overall performance on throughput, average latency, QoS between MDev-NVMe and other virtualization mechanisms. We analyse the influence of blocksizes on performance and give suggestions for NVMe virtualization to choose blocksizes in different I/O scenarios.

(5) We discuss both MDev-NVMe's performance and maintenance advantages over former virtualization And we discuss the active polling optimization in virtualization and give suggestions on polling support in the design of virtualization and storage hardware.

The rest of this paper is organized as follows. A background of NVMe storage protocol and the motivation on NVMe virtualization are provided §2. The design is demonstrated and implementation details of MDev-NVMe are detailed in §3. The results of the comparison evaluations between MDev-NVMe and other mechanisms are located within §4. There is a discussion on polling in §5. §6 details related developments, and finally the conclusion is located in §7.

2 Background and Motivation

2.1 NVMe Storage Protocol

For a long time, SATA (Serial ATA) has been used as the interface for traditional storage devices such as Hard Disk Drives (HDD) and a number of Solid State Drives(SSD). Despite the fact that SATA is enough for a rotational storage, the SATA interface cannot meet the requirement of modern storage devices. This is because of the requirement to provide a much higher I/O throughput due to the limitation of Advance Host Controller Interface (AHCI) architecture design in SATA interface. To resolve the I/O performance bottleneck, the NVMe protocol was designed and developed with a PCIe interface instead of SATA [21]. Generally, to accelerate the I/O speed of SSD, NVMe protocol optimized command issues between the host system and the storage devices, compared with the traditional ATA AHCI protocol. Concurrently, it includes support for parallel operations by supporting up to 64K commands within a single I/O queue to the device [21].

Here are some basic definitions in NVMe protocols. NVMe defines two main types of commands: **Admin Commands** and **I/O Commands**. In I/O operations, commands are placed by the host software into the **Submission Queue (SQ)**, and completion information received from SSD hardware is then placed into an associated **Completion Queue (CQ)** by the controller. NVMe separately designs SQ and CQ pairs for any Admin and I/O commands respectively. The host system maintains only one Admin SQ and its associated Admin CQ for the purpose of storage management and command control, while the host can maintain a maximum of 64K I/O SQs or CQs. The depth of the Admin SQ or CQ is 4K, where the Admin Queue can store at most 4096 entries, while the depth of I/O Queues is 64K. SQ and CQ should work in pairs, and normally one SQ utilizes on one CQ or multiple SQs utilize the same CQ to meet the requirements of high performances in multithread I/O processing. A SQ or CQ is a ring buffer and it is a memory area which is shared with the device that can be accessed by Direct Memory Access (DMA). Moreover, a doorbell is a register of the NVMe device controller to record the head or tail pointer of the ring buffer (SQ or CQ).

A specific command in a NVMe IO request contains concrete read/write messages and an address pointing to the DMA buffer if the IO request is a DMA operation. Once the request is stored in a SQ, the host writes the doorbell and kicks (transfers) the request into the NVMe device so that the device can fetch I/O operations. After an IO request has been completed, the device will subsequently write the success or failure status of the request into a CQ and the device then generates an interrupt request into the host. After the host receives the interrupt and processes the completion entries, it writes to the doorbell to release the completion entries.

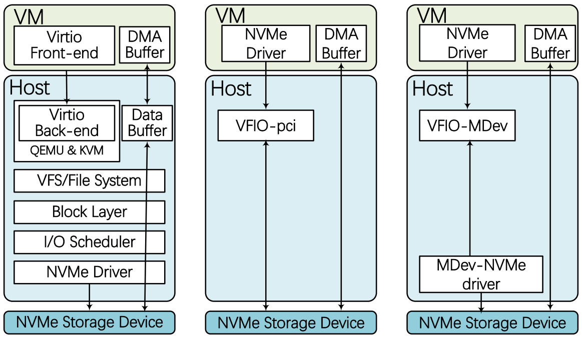
2.2 Motivation

High-performance cloud computing applications have raised great demands on the I/O performance of modern datacenters. The I/O virtualization is widely deployed in datacenter servers to support the heavy data transmission and storage tasks within cloud computing workloads. The virtualization for the new high-performance storage devices NVMe is a concentration point of I/O virtualization in cloud computing. This is because the NVMe devices can demonstrate great performance advantages over the traditional devices on the native host servers. There have been a number of previous successful NVMe virtualization solutions including VM boot disk and server side caching of VM data, both of which prove that NVMe is well suited for exploitation in various virtualization. To ensure all the VMs in cloud computing servers share the benefits of NVMe devices from the basic I/O virtualization has become essential in virtualization research for NVMe. The NVMe devices in VMs provides some basic requirements for the virtualization mechanisms:

High Throughput: The throughput performance of NVMe storage is the most important performance feature due to the requirements for both quick data access and parallel processing of cloud computing services. The virtualization mechanism needs to ensure the high performance of the NVMe devices in VMs, so the throughput-intensive services can work increasingly efficiently.

Low Latency: From previous research, the unloaded read latency of NVMe SSD storage is 20-100μs [9]. Storage virtualization mechanisms usually cause latency overhead in VMs because the high frequency of I/O operations will bring large numbers of context switches. To satisfy latency-sensitive applications in cloud computing services, NVMe virtualization should suffer a low latency overhead.

Device Sharing: Modern high-performance datecenter servers often support large numbers of VMs for separate cloud computing services. The limited NVMe storage devices should be shared by different VMs which can show great scalability. With device sharing, different VMs can work with high-performing NVMe without interference between each other, and the statuses of different VMs need to be managed by the hypervisor. Also, supporting VM live migration [32] is also an important requirement on the virtualization mechanism, which greatly relies on device sharing features. Also, the tail latency in VMs shows the QoS of device sharing among different VMs.



(a) Virtio (b) VFIO (c) MDev-NVMe

Figure 1: NVMe virtualization comparisons.

The comparisons between *Virtio* and *VFIO* and our design are displayed in Fig. 1. The previous NVMe storage virtualization *Virtio* and *Virtual Function I/O (VFIO)* [29] both have their shortcomings: *Virtio* uses the original *virtio-blk* [20] driver and it has no specific optimization for NVMe. Consequently, *Virtio* suffers readily apparent overhead from software layers which originate from guest OS to the NVMe devices, including the Virtual File System (VFS), block layer and I/O scheduler. Whereas in *Virtio*, the throughput and latency in VMs can only meet 50% of the native performance.

On the other hand, *VFIO* method can use direct pass-through [1, 10] to assign the entire NVMe device to a single VM, therefore *VFIO* can not meet the requirements of device sharing despite its near-native performance on both latency and throughput. Also, with direct pass-through, the information of virtual NVMe can not be managed by the hypervisor so it cannot support migration.

As shown in Fig. 1(c), the mediated pass-through virtualization is a full virtualization mechanism which combines both the advantages of the two former mechanisms, and it requires no modification to guest OS kernel and these guests can use the raw NVMe drivers. A “VFIO-MDev” interface is implemented for each individual VM to cooperate with QEMU [6] to support device sharing between multi VMs, which “VFIO-MDev” can emulate all the PCIe device features for each guest NVMe devices. In order to guarantee a near-native I/O performance in mediated pass-through, the direct concept of the proposed MDev-NVMe is to optimize *VFIO* by modifying the raw NVMe driver into the MDev-NVMe driver in Linux kernel, so that it can pass-through basic units of I/O operations as many as possible. As a result, all the statuses of physical and virtual queues can be managed by the MDev-NVMe module, and any file system on the host kernel is unnecessary, which helps reduce the software stack overhead. In general, a mediate pass-through is a superior choice which can meet all the essential goals of NVMe virtualization.

3 Design and Implementation

Cloud applications demand on high performance NVMe storage devices in modern datacenters. To meet both low latency and high throughput performance goals and support a sharing policy on NVMe devices, we design a mediated pass through virtualization mechanism: MDev-NVMe.

3.1 Architecture

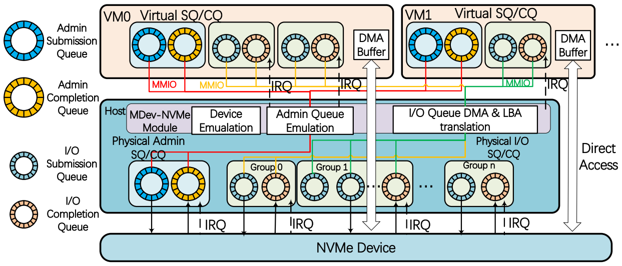


Figure 2: Architecture of MDev-NVMe.

The mediated pass-through virtualization is a full virtualization mechanism which combines both advantages of the two former mechanisms *Virtio* and *VFIO*. At the same time, it requires no modification to the guest OS kernel so the guest kernel can use the raw NVMe drivers. From the aspect of the architecture, *Virtio* provides a full device emulation with the *virtio-blk* front end driver which suffers an apparent performance overhead from software layers. And *VFIO* passes through the entire device to one VM with no sharing features. Therefore, the mediated pass-through selectively emulates or pass-throughs basic I/O units of the NVMe I/O operations. To accelerate the I/O operations and reduce the unnecessary frequent "vm-exit", we pass through I/O SQs and CQs with a queue shadow instead of I/O queue emulation. The access of DMA buffers in NVMe devices can be dealt similar to other PCIe devices.

The brief introduction of the architecture is shown in Fig. 2. From the architecture, we offer a full virtualization method, so all guests require no modification to raw NVMe drivers and can utilize all the basic NVMe commands. The MDev-NVMe is a kernel module within the Linux host system which offers three important functions: the basic PCI device emulation, Admin queue emulation and I/O queue shadow with DMA & LBA translation. Also, in MDev-NVMe, the DMA buffer in NVMe devices can be accessed by VMs as normal PCIe devices, which is similar to direct pass-through. All the guests cooperate with these three important functions.

Device Emulation: MDev-NVMe allows that all the guests need no modification to the native NVMe driver in kernels, so each guest can access a virtual NVMe device with the PCIe bus. So the MDev-NVMe module should emulate all the features of the NVMe devices for each VMs. The details of PCIe device emulation based on"VFIO-MDev" which is similar with the original "VFIO-pci": we emulated PCI registers such as BAR and other PCI configurations, and emulated NVMe registers and logic of guest device. The emulated interrupts contain INTx, MSI, and MSI-X.

Admin Queue Emulation: In NVMe protocol, there is only one pair of Admin SQ and CQ, while there can be up to 65535 pairs of I/O SQs and CQs. When different VMs generates I/O SQ and CQ at the same time, the only pair of Admin SQ and CQ in the host kernel should be able to handle the sharing and scheduling between the VMs. So we need device emulation for the Admin Queue so that all the guests can manage their NVMe I/O operations in the emulated pair of Admin Queue, and MDev-NVMe will finally parse the virtual Admin commands into physical Admin commands.

I/O Queue Shadow: Since there can be up to 64K I/O queues in a NVMe device instead of only one pair, I/O queues do not require device emulations. In our architecture, all the guest I/O commands in virtual queues can be passed through into the shadowed physical queues with the DMA & LBA translation. The guest I/O commands are directly executed by the device as the same as host commands after command translations, which can apparently reduce the overhead through emulation.

DMA Buffer Access: Similar to other PCIe devices, the host DMA engine can directly manage the memory addresses of the DMA buffer in guest NVMe storage devices, just like the *VFIO* pass-through. The DMA feature is necessary at any time that the CPU cannot maintain the rate of data transfer required, or when the CPU needs to perform work while waiting for a relatively slow I/O data transfer. The frequent I/O operations on NVMe storage devices cause a high rate of data transfer, and MDev-NVMe can overcome the CPU overhead of frequent "vm-exit" when accessing device memory.

3.2 Queue Handling

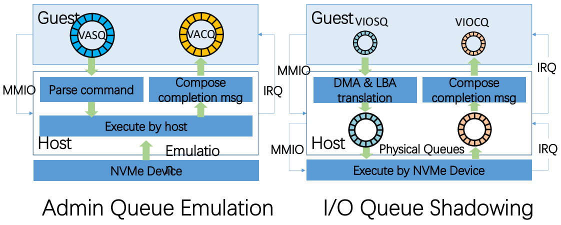


Figure 3: A brief introduction to queue handling.

We separately design queue handling mechanism for Admin Queues and I/O Queues, as shown in Fig. 3.

When more than one virtual machine runs on the host, they all maintain one pair of virtual Admin SQ and CQ in their guest OS as native. Therefore, the only physical pair of Admin SQ and CQ must be shared among the virtual machines. Concretely, MDev-NVMe usually examines the virtual Admin Queue to check the commands. If the guest Admin commands wants to create guest I/O queues, the MDev-NVMe module can directly create corresponding physical I/O queues in the host kernel. If guest Admin commands only want to check and manage the information of the I/O status and device information without attaining the concrete features and parameters of the NVMe devices. The MDev-NVMe module can directly write or update information from the NVMe doorbell register with Memory-Mapped I/O (MMIO) operation. The majority of the Admin queue operations are accomplished in the initialization process of virtual machines, so the Admin queue emulation will not become the bottleneck location of the NVMe I/O performance.

Compared with Admin queues, we use a more convenient pass-through mechanism for guest I/O SQs and CQs, where we make a queue shadow from the guest queues to the host queues. NVMe devices have more than one pair I/O SQs and CQs, so I/O queue shadow is feasible in MDev-NVMe virtualization. In host kernel, I/O queues are separately bound with physical cores of the server and interrupts of the submission and completion of each I/O commands from queues are trapped by the corresponding cores to accelerate I/O operation. To improve performance in MDev-NVMe VMs, we can also bind the guest I/O queues with the host CPU resources by shadowing guest I/O queues and host I/O queues by providing a simple DMA & LBA translation, which is introduced in the §3.3. Once the guest writes or updates an I/O command, the MDev-NVMe module will directly writes translated commands into shadowed physical queues and updates the device doorbell register with MMIO. When a guest I/O operation completes in the corresponding physical queue, the device will generate an interrupt into the host kernel, and also our MDev-NVMe module generates an interrupt into the guest kernel after checking the DMA & LBA translator. With queue shadowing, the handling of interrupts of the guest virtual queues is actually the handling of interrupts of shadowed physical queues, where we make full use of the bound CPUs (by physical queues) to accelerate the interrupt handling.

3.3 DMA & LBA translation

As we demonstrated in the architecture figure in §3.1, a guest I/O queue are shadowed with a corresponding physical I/O queue in a DMA & LBA translator to achieve resource partitioning and better I/O performance. After translation, a virtual I/O queue id are bound with a physical I/O queue id, and the translation result will be stored in a translation table which is maintained in the host kernel. The translation process is based on the Intel Extension Page Table (EPT) support for memory translation, so that we can maintain a translation table cache when multiple VMs create large numbers of virtual I/O queues and the limited physical queues must be shared.

The DMA & LBA translation are based on the static partition strategy, where all VMs are assigned with part of the continuous space of the NVMe device at the initialization process. Specifically, the translation unit in an I/O command is the data pointer which points to a DMA buffer. The address is a Guest Physical Address (GPA). The MDev-NVMe module use *vfio_pin_pages* to translate the GPA to HPA (Host Physical Address) and pin the DMA buffer in host memory. Specifically, the *vfio_pin_pages* can ensure the isolation of the guest memory between different virtual machines. Since all the translations are under control of the translator in MDev-NVMe module, a malicious VM cannot access the physical I/O queues which is not assigned to itself and can not access the I/O queues in other VMs either. The command buffer is also protected by the EPT, which helps to ensure the security of DMA buffers of different VMs. Another important unit of the I/O operation is the Start Logical Block Address (SLBA). Since our MDev-NVMe virtualization takes a static partition mechanism, the SLBA in guest I/O command can be modified and subsequently copied into the host I/O queue by applying a space area offset, which are also managed by MDev-NVMe module in host kernel.

3.4 2-Way Polling Mode

With the mediated pass-through mechanism, the NVMe devices can achieve a performance better than *Virtio*. However, the performance in VMs still show an obvious gap compared with the native platform in experiment observation. The I/O bottleneck results from frequent submissions and completions of guest I/O commands, so we discuss the detailed origin of overhead in Fig. 4.

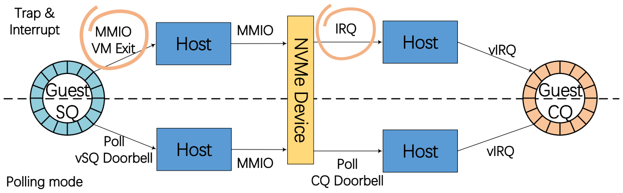


Figure 4: A guest I/O operation process in two modes.

Fig. 4 shows entire processes of a guest NVMe I/O command operation in "non-polling" or "polling" mode. Generally, the submission latency from the host kernel to the physical NVMe device is inappreciable, which is less than 10 μs in our observations from experiments. However, the guest I/O commands suffers seriously from two part overhead: (1) when the guest I/O commands are translated and submitted to the corresponding physical queues through an MMIO operation it will result in a "vm-exit", (2) when handling interrupts generated by physical device when updating a doorbell register with a completion of physical I/O queue.

To overcome the two-part overhead, a direct idea is to change the trap of interrupts into an active polling mechanism in the 2-way overhead. First, the shadowed doorbell of guest SQ or CQ can be stored in host memory area so that MDev-NVMe can directly manage all the guest I/O operations and update the NVMe queue status. With polling, we polls guests to write into the shadow doorbells instead of generating an MMIO into the host. Also, we disables interrupt integration of host CQs into guests and adopts polling new completion entries from the host CQs. As a result, the shadow SQ tail doorbell, the shadow CQ head doorbell, and the host CQ are all stored in the host memory, and they can be directly fetched and immediately updated by the polling threads.

Now in our MDev-NVMe, we use 3 threads for individually polling the shadow SQ tail doorbells, the shadow CQ head doorbells, and the host CQs so that we can get even better performance of I/O queue handling than native platform in the host kernel, which is shown in §4, while using 1 thread with round-robin polling of multiple queues cannot provide corresponding performance. The threads will bring 100% usage of 3 cores on the host server. And the polling threads can be shared between VMs.

4 Experiments

To demonstrate the I/O performance of virtual machines of our mediated pass-through virtualization mechanism, we run fio [4] I/O benchmark experiments in Linux virtual machines based on the comparison between different virtualization mechanisms, including *Virtio*, famz userspace driver, SPDK *vhost-scsi*, and SPDK *vhost-blk*. The performance of MDev-NVMe and the other 4 mechanisms are compared and normalized with the native performance on the physical devices. For Throughput performance, MDev-NVMe shows the best bandwidth performance among all the virtualization mechanisms and can provide up to 1.5× native performance. For Latency performance, MDev-NVMe presents the lowest average latency, and can give a bounded maximum latency with a reliable QoS performance. Also, the tail latency performance of MDev-NVMe is also outstanding among all the comparison experiments. In the meanwhile, our MDev-NVMe scales well without a visible performance drop.

4.1 Configuration

The evaluation concentrates on achieving the I/O performance of NVMe Storage devices and demonstrating the advantages and disadvantages of different virtualization methods, including *Virtio*, Famz userspace driver, SPDK *vhost-scsi*, and SPDK *vhost-blk*. Experiments on different concrete NVMe SSD products show different results and give us more insights. Nowadays, the most advanced NVMe Storage Device is based on Intel OPTANE Technology with 3D XPoint Memory media. The OPTANE SSD has an amazing performance which can support up to 550K IOPS in 4K random read and 500k IOPS in 4K random write with a 10μs latency. So we first build our experiment environment on the OPTANE SSD DC P4800X 375G NVMe SSD, and the server hardware platform includes 2 Intel Xeon CPU E5-2699 v3 with 18 CPU cores (2.3GHz), 64GB system memory. Besides, we also do the same experiments on a more commonly used NVMe SSD: INTEL SSD DC P3600 400G, running on the server which includes 2 Intel Xeon CPU E5-2680 v4 with 14 CPU cores (28 threads) (2.4GHz), 64GB system memory.

We run 64bit Ubuntu 16.04 with a 4.10.0 kernel with our MDev-NVMe kernel module in the host server, and 64bit Debian 9.0 with a 4.10.0 kernel in the guests. Each virtual machine is allocated with 4 VCPUs and 8GB system memory. To get the best I/O performance in comparison experiments, we set up 1-VM case which the single virtual machine can get the entire volume. To discuss the scalability of MDev-NVMe sharing, we partition the NVMe SSD with 4 60G area into 4 individual VMs.

We use the flexible I/O tester (FIO) [4] as our application evaluation benchmark. FIO is a typical I/O benchmark with different parameters to demonstrate the IO performance, and it is widely used in research and industries. Specifically, we take **libaio** as the fio engine and we run 5 groups of test scripts in Table 1, including the random read or write with "iodepth=1", the random read or write with "iodepth=32", and the 70% random read and 30% random write. The FIO test doesn't use any file system and chooses "O_DIRECT" parameter for Direct I/O experiments.

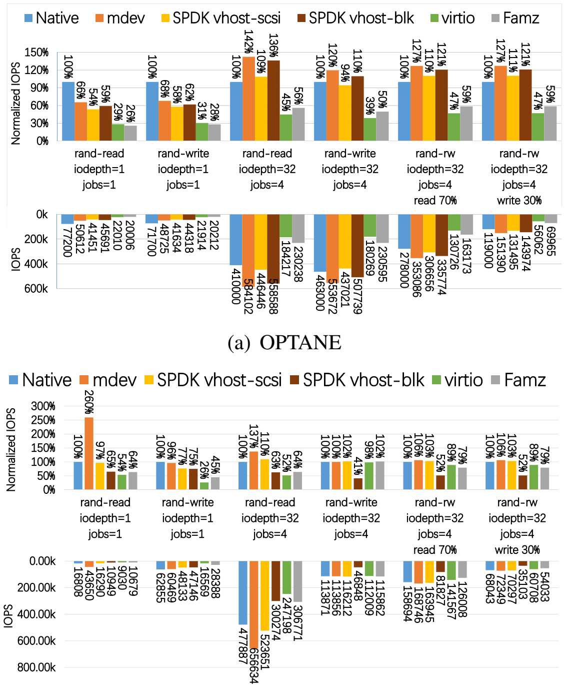
Table 1: Fio test cases

Test case	Description
rand-read-qd1	4K random read, iodepth=1, numjobs=1
rand-write-qd1	4K random write, iodepth=1, numjobs=1
rand-read-qd32	4K random write, iodepth=32, numjobs=4
rand-write-qd32	4K random write, iodepth=32, numjobs=4
rand-rw-qd32 read 70%	4K random read, iodepth=32, numjobs=4
rand-rw-qd32 write 30%	4K random write, iodepth=32, numjobs=4

4.2 Throughput Performance

We first concentrate on the throughput performance of I/O operations on OPTANE and P3600. The basic throughput benchmark concentrates on the 4K page random read and write. Fig. 5(a) and Fig. 5(b) are the corresponding IOPS results on the OPTANE and P3600 NVMe SSDs. The results in these figures contain two parts: the upper part is the normalized performance results based on the native performance baseline, and the bottom part is the original benchmark IOPS results. On OPTANE, the MDev-NVMe mechanism shows the best throughput performance with or without I/O multi-queue optimization(“iodepth=1” or “iodepth=32”) over all the other virtualization mechanisms. With the active polling for I/O queues, MDev-NVMe can make full use of multi-queue features in 4K random read and write I/O benchmarks with “iodepth=32”, where MDev-NVMe shows up to 142% performance over native results. Since the SPDK *vhost-scsi* and SPDK *vhost-blk* mechanisms utilize similar idea of polling, they can also achieve better performance than native results. On P3600 NVMe SSDs, MDev-NVMe show the best performance although the advantages are not as obvious as the OPTANE experiments. Moreover, the throughput performance of 4K random write with multi-queue optimization presents an apparent gap compared with the 4K random read with multi-queue optimization, as we shown in the bottom part of Fig. 5(b).

In this group of experiments, different virtualization mechanisms show well-matched performance expect SPDK *vhost-blk*, which shows that SPDK *vhost-blk* optimization does not works efficiently on P3600.



(a) OPTANE

(b) P3600

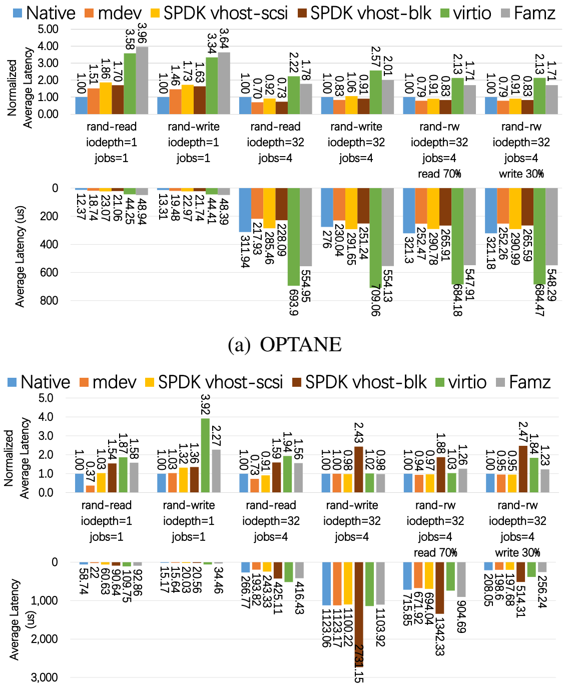
Figure 5: Throughput performance of IOPS.

4.3 Latency Performance

We also focus on the latency of I/O operations in virtual machines based on different NVMe virtualization mechanisms. We particularly discussed the average latency, max latency and the tail latency on different virtualization mechanisms in our latency experiments. Specifically, the latency measured in our fio experiments contains both submission latency and completion latency.

4.3.1 Average Latency

Firstly, the average of latency performance in benchmarks are demonstrated in Fig. 6(a) and Fig. 6(b). Also the upper part is the normalized performance results based on the native performance baseline, and the bottom part is the original latency results. The average latency performance can show the average operation overhead of each I/O commands in NVMe devices through the software stack of virtualization.



(a) OPTANE

(b) P3600

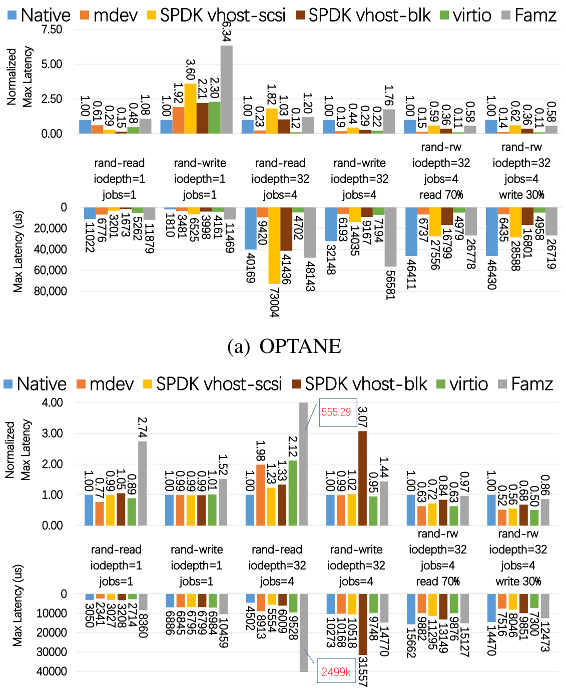
Figure 6: Average latency performance.

As we talked in the former section, the mediated pass-through takes advantage of the *VFIO* pass-through mechanisms and it overcomes the block layer and Backend software stack. When there is no multi-queue optimization [8, 18], all the virtualization mechanism shows no apparent advantage over the native latency performance. The MDev-NVMe, SPDK *vhost-scsi* and SPDK *vhost-blk* can show 2× latency performance over the *Virtio* and Famz userspace driver, and the original latency intervals between native performance and MDev-NVMe is nearly 10 μs, which is at the level of the latency from the host to physical NVMe devices. In multi-queue experiments where “iodepth=32”, the native latency increase to 322 μs which is about 30× the result with “iodepth=1”. However, in multi-queue benchmarks, Mdev-NVMe show the best performance and can achieve only 81% of the native performance. MDev-NVMe shows over 2.5× advantages over *Virtio* and Famz drivers, and the advantages over SPDK *vhost-scsi* and SPDK *vhost-blk* are not far but obvious. On P3600, MDev-NVMe can show great results on all the test cases, except the random write with "iodepth=32" benchmark where all the virtualization performance results are similar. Similar to the throughput benchmarks, SPDK *vhost-blk* does not show the optimization results as those on OPTANE.

In general, MDev-NVMe can achieve the best average latency performance, so our mediated pass-through mechanism has overcome utmost software layer I/O overhead.

4.3.2 QoS

To talk about the QoS performance of the NVMe devices in virtual machines, we concentrate on the maximum latency and tail latency performance in different benchmark test cases.

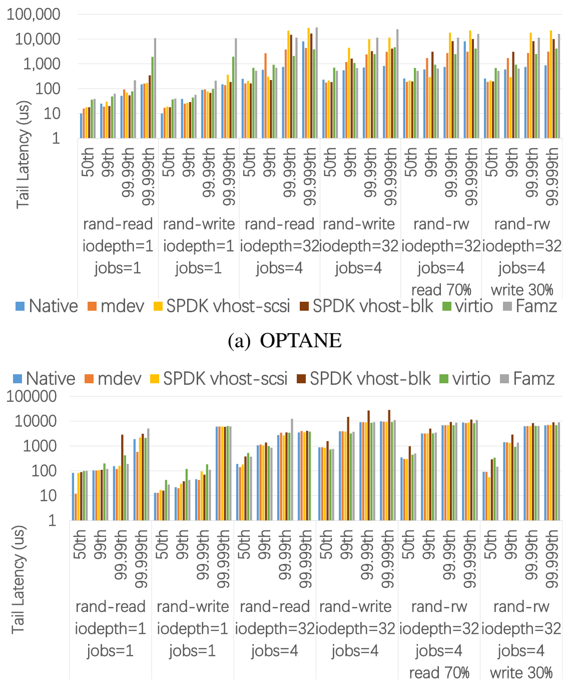


(a) OPTANE

(b) P3600

Figure 7: Maximum Latency performance.

Firstly, the max of latency performance in benchmarks is demonstrated in Fig. 7(a) and Fig. 7(b). For Qos performance, our MDev-NVMe shows well-matched performance of the maximum latency with *Virtio* on both OPTANE and P3600. Also, the normalized maximum latency performance of MDev-NVMe can be less than 1, showing the latency performance can be bound in a very low level. In this part of experiments, SPDK *vhost-scsi* and SPDK *vhost-blk* can not offer a well-matched performance with *Virtio* and MDev-NVMe on both Optane and P3600. Specially, Famz userspace NVMe driver cannot bound its maximum latency in an acceptable level because in Fig. 7(b), the maximum latency of random read with "iodepth=32" is 2499k μs. Also, SPDK *vhost-blk* present a straggler QoS performance on P3600 when doing the random write with "iodepth=32" benchmarks.



(a) OPTANE

(b) P3600

Figure 8: Tail Latency performance.

And we also present the tail latency performance in Fig. 8(a) and Fig. 8(b) with a logarithmic scale coordinate of the latency results. From the figures, MDev-NVMe can bind the 50th and 99th latency performance in similar level. We also compare 99th, 99.99th and 99.999th tail latency in different virtualization mechanisms. MDev-NVMe has the smallest intervals between 99th, 99.99th and 99.999th latency compared with other virtualization mechanisms. In general, MDev-NVMe can provide a promising QoS performance of NVMe virtualization.

4.4 Scalability

Our mediated pass-through mechanism can provide device sharing over a NVMe SSD on the initialization process of virtual machines with a static partition of the device. We design an experiment by separately running 1 VM, 2VMs, and 4VMs and evaluating the virtual NVMe I/O performance to discuss the scalability. In our experiment, each VM manages a 60G continuous storage area by the unmodified guest driver. Fig. 9 presents the relative scalability performance results of MDev-NVMe on the P3600 NVMe device with the scripts in Table 1, and each result is the sum of all throughput in all VMs. As we talked about in the former section, the random write performance is not well-matched with the random read performance. When increasing VMs in the benchmark with "iodepth=1", the performance of 4K random read can increase by the numbers of VMs because the NVMe read performances are not fully exerted when there is no multi-queue optimization [8, 18]. In the other test cases, since the I/O queue resource has been fully used, the I/O performance stays in a stable status when VMs increase from 1 to 4. Since the basic units of NVMe I/O operation are the SQs and CQs, and the performance in VMs is strictly connected with the number of queues assigned for them. As a result, when the number of VMs increases, the assignment of limited I/O queues becomes the bottleneck of scalability. When the queue depth is small, setting up more VMs can utilize make better use of queue resource. When the queue depth is large, MDev-NVME can guarantee that multi VMs can work together without performance obvious drop. So we can conclude that MDev-NVME can ensure a promising scalability.

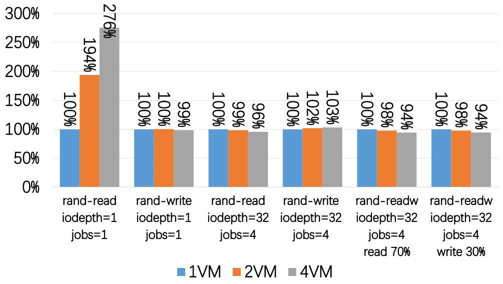


Figure 9: IOPS performance of multi VMs on P3600.

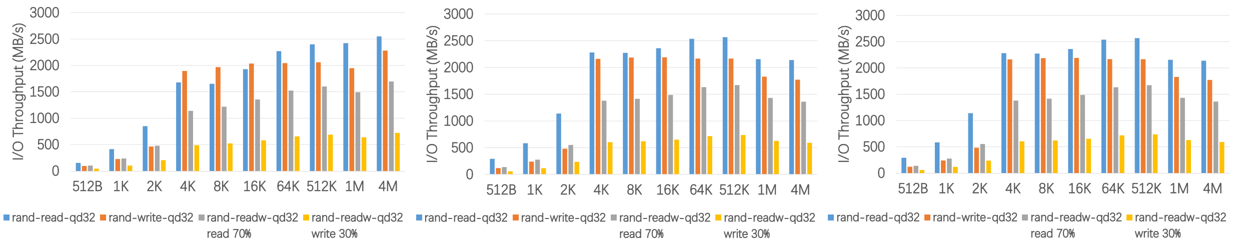
4.5 Influence of I/O Blocksize

The former experiments are based on the 4K random read or write benchmarks, which are concentrated points of test cases on different storage devices. Moreover, the I/O blocksize can explicitly influence the performance of devices wherever on the native devices or in virtual machines. As a result, we want to take the blocksize parameter of I/O performance into consideration to give a law of performance varying with different blocksizes, and we can also give more suggestions for different NVMe virtualization mechanisms to make a right choice of blocksizes in different I/O scenarios.

We run the fio test cases with multi-thread optimization ("iodepth=32") which are the last four test cases presented in Table 1. We chose 10 groups of blocksize parameters in each group of random read of write experiments on OPTANE (512B, 1K, 2K, 4K, 8K, 16K, 64K, 512K, 1M, 4M). The results of the influence of blocksizes on bandwidths are presented in Fig. 10(a), 10(b) ,and 10(c). We only chose the native, MDev-NVMe, and SPDK *vhost-blk* experiments as a comparison since MDev-NVMe and SPDK *vhost-blk* show the best 4K I/O performance based on our prior experiments on OPTANE. In Fig. 10, the variation trends of the influence of blocksizes on bandwidths in the three sub figures basically agree with each other. When the blocksizes are smaller than 4K, the throughput performance improve with the blocksize parameter and the high throughput performance are not fully exerted. These is because the NVMe I/O queues can be 64K large at most and when the blocksize is too small and the device is not fully occupied, resulting in a I/O queue resource waste. Also the throughput performance can be continuously improved with the growing blocksize on native devices, but the performance results of MDev-NVMe and SPDK shows apparent declines when the blocksize is larger than 512K. The reason for this performance drop is that when the I/O blocksize is too large, then transmission of an I/O block may be automatically separated by the hypervisor and these separations bring an appreciable overhead. A single I/O command of such a large block needs cooperation of several I/O queues and the scheduling of queues will become a performance bottleneck.

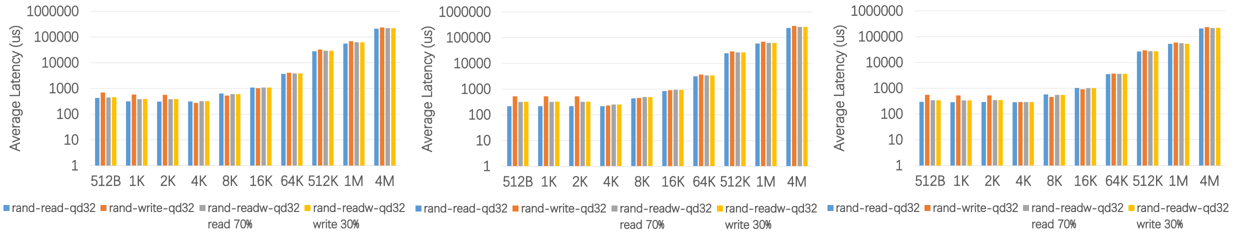
Also, we demonstrate the average latency performance of native, MDev-NVMe, and SPDK *vhost-blk* experiments in Fig. 11(a), 11(b), and 11(c). We use the a logarithmic scale coordinate to present the average latency experiment results in the three sub figures from Fig. 11. The variation trends of the influence of blocksizes on average latency performance in the three sub figures agree with each other as well as the throughput performance results. The average latency slightly decreases from 512B to 4K and then progressively increases from 4K to 4M. The MDev-NVME and SPDK can all bind their average latency performance in the same order of magnitude as the native performance. When the blocksize $\leq$ 16K, the average latency performance is in the level of several hundred μs. When the blocksize is chosen as 64K, 512K, 1M and 4M, the average latency performance increases exponentially by the magnitude. The main reason for this unacceptable latency are mainly resulted by the I/O block separations and the overhead waiting for all queue competitions.

In general, the throughput performance is good when blocksizes is bigger than 4K while the average latency performance will become unacceptable when the blocksize is bigger than 64K. Our MDev-NVMe can provide a stable and excellent performance of both throughput and latency when choosing blocksizes as 4K, 8K, and 16K, and the performance is better than SPDK and native performance. To support more optional blocksize with the high performance, we would give a more efficient I/O queue scheduling in our future works.



(a) Native performance (b) MDev-NVMe performance (c) SPDK performance

Figure 10: The throughput of I/O bandwidth on different blocksizes on OPTANE NVMe device.



(a) Native performance (b) MDev-NVMe performance (c) SPDK performance

Figure 11: The average latency of I/O operation on different blocksizes on OPTANE NVMe device.

5 Discussions

In this section we discuss the advantages of MDev-NVMe over other viruralization mechanisms, the overhead issues, and the importance of active polling in virtualization.

Advantages of MDev-NVMe: As we demonstrated in §4, MDev-NVMe presents outstanding performance on throughput, average latency, and QoS. Furthermore, the data from the experiments suggests the performance in VMs can be even better than a native device. In our comparison experiment, MDev-NVMe can achieve over 2.5× throughput performance of Virtio and Famz, and achieve less than 31% average latency of *Virtio* and Famz. SPDK *vhost-scsi* and *vhost-blk* can provide a promising performance, slightly inferior to MDev-NVMe. However, SPDK needs Hugepage memory and additionally the setting up of a SPDK *vhost* device on the NVMe device, so it may restrict the flexibility of physical NVMe devices. Our MDev-NVMe is implemented as a kernel module, offering more convenience for cloud administrators and users with its better maintenance than SPDK, and MDev-NVMe brings no memory overhead.

Overhead issues: The shadow of guest I/O queues to physical queues is determined by the MDev-NVMe module. The scheduling of I/O queues is based on a simple FIFO policy. When aggressive I/O tasks in VMs are competing for the limited physical queues it can increase the scheduling overhead in the host and will lead to some problems in the balancing of requests. Here, a increasingly complicated scheduling algorithm is implemented or a queue resource ballooning methodology could be used in future work to overcome the potential overhead in heavy I/O workloads. To overcome the I/O performance bottleneck results from the high frequency vm-exit of guest I/O commands, an additional CPU resource is used for polling in order to accelerate the interrupt operations. A polling thread aggressively utilizes the CPU resource of a single core, which transfers overhead to the host server when running large numbers of VMs. However, the proposed system ensures no CPU resource overhead to VMs since all the guests use native NVMe drivers which are not aware of the polling threads. Moreover, polling can significantly reduce CPU usage in the VM because it avoids "vm-exit".

Importance of polling: The main agreement of MDev-NVMe and SPDK is that the virtualization mechanisms take active polling instead of interrupts handling. The traditional virtualization mechanisms are often based on the trap and emulations. The trap of the "vm-exit" needs additional context switches between VMs and the host kernel. The NVMe protocol is an inherently parallel and high-performing interface and command set. Therefore, when we would like to make full use of the I/O queue resource, the large numbers of interrupts bring an appreciable overhead for guests. Now, assigning exclusive CPU computing resources to handle the high frequency of interrupts can directly help to achieve high performance of I/O operations in VMs. Taking polling is the most direct and obvious idea to assign the exclusive CPU resource for queue handling.

Despite the high utilization of polling CPU may generate limited CPU computing resource waste to the host servers, polling is still necessary when the I/O workloads are aggressive in achieving extremely high performance.

For example, when shopping seasons such as "Black Friday" arrive, the servers of e-commerce companies will face great pressure of database accesses and parallel processing. Subsequently, the storage device throughput and latency performance is directly connected with the respective companies profits, losses and their customers satisfaction. Supporting polling in the virtualization mechanism is essential to take full advantage of the benefits of NVMe devices to meet the requirements on I/O performance in datacenters. Moreover, NVMe virtualization should support adaptive polling, which allows such datacenter administrators to decide when to choose a mild policy for I/O acceleration in VMs, to reduce expenses and support an increasing number of VMs. Therefore the provision of an adaptive polling mode and increasingly optimized polling algorithms is part of our focus for future work. It is also expected that the high performance I/O device hardware will be designed with a number of components to actively support or cooperate with the polling algorithm in the near future.

6 Related Work

Some research concentrates on NVMe virtualization, including the para-virtualization abstraction *Virtio*, a userspace NVMe driver in Qemu, and the Storage Performance Development Kit (SPDK).

VMware virtual NVMe device: After vSphere 6.5, VMware adds an NVMe controller for NVMe devices in its virtualization solution. The biggest benefit of using an NVMe interface over the traditional SCSI is that it can significantly reduce the amount of overhead, as well as reducing the IO latency for VM workloads.

Virtio for NVMe: *Virtio* is an abstraction for a set of common emulated devices in a para-virtualized hypervisor, which allows the hypervisor to export a common set of emulated devices and make them available through the costumed API. Briefly, *Virtio* is easy in the implementation of storage device virtualization. *Virtio* inevitably increases the I/O path which indicates that guest I/O request goes through both guest and host I/O path. Data replication between guest and host can also have an impact on performance.

Userspace NVMe Driver in QEMU: Fam Zheng [34] implements a NVMe driver with *VFIO* as the driver of NVMe device to cooperate with the modified userspace NVMe driver in Qemu. Compared with the *VFIO* method, this userspace driver emulates several software layer, that is the BlockBackend, Block layer, and Qcow2 layers in Qemu. This userspace NVMe driver takes advantages of *VFIO*, enabling NVMe device to gain more I/O software layer features at the cost of device sharing capability and this virtualization mechanism brings apparent latency to the VMs.

SPDK: The Storage Performance Development Kit (SPDK) is a user-mode application which aims at providing a high performance and scalable I/O application interfaces for different storage devices. It integrates all of the necessary drivers into userspace and operates with an enforced polling mode to achieve high I/O performances by taking advantage of DPDK [14]. Specifically, SPDK application extends "SPDK vhost" to present *Virtio* storage controllers to QEMU-based VMs with *vhost-scsi* and *vhost-blk* [20] interfaces.

7 Conclusion

Within this paper, there is introduction of a new virtualization implementation for NVMe storage device MDev-NVMe. The MDev-NVMe is a full NVMe storage virtualization mechanism where all the VMs can run native NVMe driver. The proposed solution takes a mediated pass-through as the main implementation containing the Admin queue emulations and I/O queues shadowing on the basis of Device emulation for all PCI configurations. To achieve the most outstanding performance of the NVMe device among mainstream NVMe virtualization mechanisms, it is proposed that an active polling mode is implemented, which can achieve both high throughput, low latency performance and a reliable device scalability on both Intel OPTANE and P3600 NVMe SSD. With the MDev-NVMe module, the physical device can be partitioned to support device sharing, and each VM can own a full-fledged NVMe device which can directly access the physical performance-critical resources without interference with other VMs. The large numbers of Fio benchmark experiments provides evidence for the great performance of MDev-NVMe, which is better than native performance and other existing virtualization mechanisms. There is also focus and consideration into the influence of blocksize on the I/O performance to provide more suggestions to different I/O applications. Finally, we focus on the high performance of MDev-NVMe by analysis of the performance comparisons between different mechanisms, raising a discussion about polling. Therefore recommendations are provided for the optimization on polling support from both aspects for the design of storage hardware and also, virtualization. In the future, further research on the support for polling will be conducted as well as the optimization of the polling algorithm for MDev-NVMe.

8 ACKNOWLEDGMENT

This work was supported in part by National Key Research & Development Program of China (No. 2018YFB1003603), the Program for NSFC (No. 61772339, 61525204), and the Shanghai Rising-Star Program (No.16QA1402200). The corresponding author is Prof. Yao, Jianguo.

References

[1] Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., and Wiegert, J. Intel virtualization technology for directed i/o. *Intel technology journal 10*, 3 (2006).

[2] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Disk-locality in datacenter computing considered irrelevant. In *HotOS* (2011), vol. 13, pp. 12–12.

[3] Awad, A., Kettering, B., and Solihin, Y. Non-volatile memory host controller interface performance analysis in high-performance i/o systems. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on* (2015), IEEE, pp. 145–154.

[4] Axbo, J. Fio: Flexible i/o tester. `https://github.com/axboe/fio`, 2015.

[5] Barroso, L. A., Clidaras, J., and Hölzle, U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture 8*, 3 (2013), 1–154.

[6] Bellard, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), vol. 41, p. 46.

[7] Bergman, S., Brokhman, T., Cohen, T., and Silberstein, M. Spin: Seamless operating system integration of peer-to-peer dma between ssds and gpus. In *Proceedings of the Seventeenth USENIX Annual Technical Conference, USENIX ATC* (2017), vol. 17.

[8] Bjørling, M., Axboe, J., Nellans, D., and Bonnet, P. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference* (2013), ACM, p. 22.

[9] Colombani, F. A. Hdd, sshd, ssd or pcie ssd. `http://www.storagenewsletter.com/rubriques/market-reportsresearch/hdd-sshd-ssd-or-pcie-ssd/`, 2015.

[10] Dong, Y., Dai, J., Huang, Z., Guan, H., Tian, K., and Jiang, Y. Towards high-quality i/o virtualization. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), ACM, p. 12.

[11] Dong, Y., Yu, Z., and Rose, G. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O Virtualization* (2008), vol. 2.

[12] Dowty, M., and Sugerman, J. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 73–82.

[13] Huynh, K. Exploiting the latest kvm features for optimized virtualized enterprise storage performance. *CloudOpen, North America* (2013).

[14] Intel. Dataplane performance development kit. `http://dpdk.io/`.

[15] Intel. Storage performance development kit. `http://www.spdk.io/`.

[16] Keeriyadath, S. Nvme virtualization ideas for machines on cloud. `http://www.snia.org/sites/default/files/SDC/2016/presentations/solid\_state\_storage/Sangeeth\_NVMe\_Virtualization\_Ideas.pdf`, 2016.

[17] Kim, J., Ahn, S., La, K., and Chang, W. Improving i/o performance of nvme ssd on virtual machines. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (2016), ACM, pp. 1852–1857.

[18] Kim, T. Y., Kang, D. H., Lee, D., and Eom, Y. I. Improving performance by bridging the semantic gap between multi-queue ssd and i/o virtualization framework. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on* (2015), IEEE, pp. 1–11.

[19] Klimovic, A., Kozyrakis, C., Thereska, E., John, B., and Kumar, S. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 29.

[20] Lei, M. Virtio-blk multi-queue conversion and qemu optimization. `http://events.linuxfoundation.org/sites/events/files/slides/virtio-blk_qemu_v0.96.pdf`.

[21] nvmexpress. Nvm express specification. `http://www.nvmexpress.org/specifications/`.

[22] Ouyang, J., Lin, S., Jiang, S., Hou, Z., Wang, Y., and Wang, Y. Sdf: software-defined flash for web-scale internet storage systems. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, ACM, pp. 471–484.

[23] Russell, R. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review 42*, 5 (2008), 95–103.

[24] Son, Y., Han, H., and Yeom, H. Y. Optimizing file systems for fast storage devices. In *Proceedings of the 8th ACM International Systems and Storage Conference* (2015), ACM, p. 8.

[25] Suzuki, Y., Kato, S., Yamada, H., and Kono, K. Gpuvm: Why not virtualizing gpus at the hypervisor? In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), pp. 109–120.

[26] Technologies, A. Storage and pci express a natural combination. `http://www.avagotech.com/applications/datacenters/enterprise-storage`, 2016.

[27] Tian, K., Dong, Y., and Cowperthwaite, D. A full gpu virtualization solution with mediated pass-through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), pp. 121–132.

[28] Vučinić, D., Wang, Q., Guyot, C., Mateescu, R., Blagojević, F., Franca-Neto, L., Le Moal, D., Bunker, T., Xu, J., Swanson, S., et al. Dc express: shortest latency protocol for reading phase change memory over pci express. In *Proceedings of the 12th USENIX conference on File and Storage Technologies* (2014), USENIX Association, pp. 309–315.

[29] Williamson, A. Vfio: A user's perspective. `https://www.linux-kvm.org/images/b/b4/2012-forum-VFIO.pdf`, 2012.

[30] Xia, L., Lange, J., Dinda, P., and Bae, C. Investigating virtual passthrough i/o on commodity devices. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 83–94.

[31] Xu, Q., Siyamwala, H., Ghosh, M., Suri, T., Awasthi, M., Guz, Z., Shayesteh, A., and Balakrishnan, V. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference* (2015), ACM, p. 6.

[32] Zhai, E., Cummings, G. D., and Dong, Y. Live migration with pass-through device for linux vm. In *OLS08: The 2008 Ottawa Linux Symposium* (2008), pp. 261–268.

[33] Zhang, Y., and Swanson, S. A study of application performance with non-volatile main memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on* (2015), IEEE, pp. 1–10.

[34] Zheng, F. Userspace nvme driver in qemu. `https://events.static.linuxfound.org/sites/events/files/slides/Userspace%20NVMe%20driver%20in%20QEMU%20-%20Fam%20Zheng_0.pdf`.

AutoSSD: an Autonomic SSD Architecture

Bryan S.Kim
Seoul National University

Hyun Suk Yang
Hongik University

Sang Lyul Min
Seoul National University

Abstract

From small mobile devices to large-scale storage arrays, flash memory-based storage systems have gained a lot of popularity in recent years. However, the uncoordinated use of resources by competing tasks in the flash translation layer (FTL) makes it difficult to guarantee predictable performance.

In this paper, we present *AutoSSD*, an autonomic SSD architecture that self-manages FTL tasks to maintain a high-level of QoS performance. In AutoSSD, each FTL task is given an illusion of a dedicated flash memory subsystem, allowing tasks to be implemented oblivious to others and making it easy to integrate new tasks to handle future flash memory quirks. Furthermore, each task is allocated a share that represents its relative importance, and its utilization is enforced by a simple and effective scheduling scheme that limits the number of outstanding flash memory requests for each task. The shares are dynamically adjusted through feedback control by monitoring key system states and reacting to their changes to coordinate the progress of FTL tasks.

We demonstrate the effectiveness of AutoSSD by holistically considering multiple facets of SSD internal management, and by evaluating it across diverse workloads. Compared to state-of-the-art techniques, our design reduces the average response time by up to 18.0%, the 3 nines (99.9%) QoS by up to 67.2%, and the 6 nines (99.9999%) QoS by up to 76.6% for QoS-sensitive small reads.

1 Introduction

Flash memory-based storage systems have become popular across a wide range of applications from mobile systems to enterprise data storages. Flash memory's small size, resistance to shock and vibration, and low power consumption make it the *de facto* storage medium in mobile devices. On the other hand, flash memory's low latency and collectively massive parallelism make flash storage suitable for high-performance storage for mission-critical applications. As multi-level cell technology [5] and 3D stacking [38] continue to lower the cost per GB, flash storage will not only remain competitive in the data storage market, but also will enable the emergence of new applications in this *Age of Big Data*.

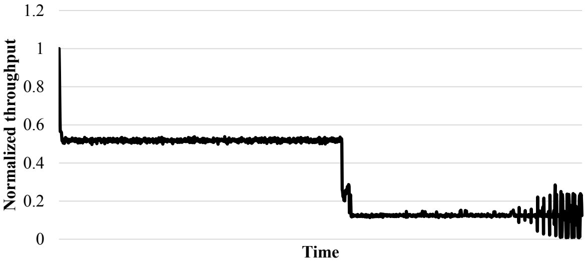


Figure 1: Performance drop and variation under 4KB random writes.

Large-scale deployments and user experiences, however, reveal that despite its low latency and massive parallelism, flash storage exhibits high performance instabilities and variations [9, 17]. Garbage collection (GC) has been pointed out as the main source of the problem [9, 25, 28, 29, 45], and Figure 1 illustrates this case. It shows the performance degradation of our SSD model under small random writes, and it closely resembles measured results from commercial SSDs [2,23]. Initially, the SSD's performance is good because all the resources of the flash memory subsystem can be used to service host requests. But as the flash memory blocks are consumed by host writes, GC needs to reclaim space by compacting data spread across blocks and erasing unused blocks. Consequently, host and GC compete for resources, and the host performance inevitably suffers.

However, garbage collection is a necessary evil for the flash storage. Simply putting off space reclamation or treating GC as a low priority task will lead to larger performance degradations, as host writes will eventually block and wait for GC to reclaim space. Instead, garbage collection must be judiciously scheduled with host requests to ensure that there is enough free space for future requests, while meeting the performance demands of current requests. This principle of harmonious coexistence, in fact, extends to every internal management task. Map caching [15] that selectively keeps mapping data in memory generates flash memory traffic on cache misses, but this is a mandatory step for locating host data. Read scrubbing [16] that preventively migrates data before its corruption also creates traffic when blocks are repeatedly read, but failure to perform its duty on time can lead to data loss. As more tasks with unique responsibilities are added to the system, it becomes increasingly difficult to design a system that meets its performance and reliability requirements [13].

In this paper, we present an autonomic SSD architecture called AutoSSD that self-manages its management tasks to maintain a high-level of QoS performance. In our design, each task is given a virtualized view of the flash memory subsystem by hiding the details of flash memory request scheduling. Each task is allocated a share that represents the amount of progress it can make, and a simple yet effective scheduling scheme enforces resource arbitration according to the allotted shares. The shares are dynamically and automatically adjusted through feedback control by monitoring key system states and reacting to their changes. This achieves predictable performance by maintaining a stable system. We show that for small read requests, AutoSSD reduces the average response time by up to 18.0%, the 3 nines (99.9%) QoS by up to 67.2%, and the 6 nines (99.9999%) QoS by up to 76.6% compared to state-of-the-art techniques. Our contributions are as follows:

- We present AutoSSD, an autonomic SSD architecture that dynamically manages internal housekeeping tasks to maintain a stable system state. (§ 3)
- We holistically consider multiple facets of SSD internal management, including not only garbage collection and host request handling, but also mapping management and read scrubbing. (§ 4)
- We evaluate our design and compare it to the state-of-the-art techniques across diverse workloads, analyze causes for long tail latencies, and demonstrate the advantages of dynamic management. (§ 5)

The remainder of this paper is organized as follows. § 2 gives a background on understanding why flash storages exhibit performance unpredictability. § 3 presents the overall architecture of AutoSSD and explains our design choices. § 4 describes the evaluation methodology and the SSD model that implements various FTL tasks, and § 5 presents the experimental results under both synthetic and real I/O workloads. § 6 discusses our design in relation to prior work, and finally § 7 concludes.

2 Background

For flash memory to be used as storage, several of its limitations need to be addressed. First, it does not allow in-place updates, mandating a mapping table between the logical and the physical address space. Second, the granularities of the two state-modifying operations—program and erase—are different in size, making it necessary to perform garbage collection (GC) that copies valid data to another location for reclaiming space. These internal management schemes, collectively known as the flash translation layer (FTL) [11], hide the limitations of flash memory and provide an illusion of a traditional block storage interface.

The role of the FTL has become increasingly important as hiding the error-prone nature of flash memory can be challenging when relying solely on hardware techniques such as error correction code (ECC) and RAID-like parity schemes. Data stored in the flash array may become corrupt in a wide variety of ways. Bits in a cell may be disturbed when neighboring cells are accessed [12, 41, 44], and the electrons in the floating gate that represent data may gradually leak over time [6, 35, 44]. Sudden power loss can increase bit error rates beyond the error correction capabilities [44, 47], and error rates increase as flash memory blocks wear out [6, 12, 19]. As flash memory becomes less reliable in favor of high-density [13], more sophisticated FTL algorithms are needed to complement existing reliability enhancement techniques.

Even though modern flash storages are equipped with sophisticated FTLs and powerful controllers, meeting performance requirements have three main challenges. First, as new quirks of flash memory are introduced, more FTL tasks are added to hide the limitations, thereby increasing the complexity of the system. Furthermore, existing FTL algorithms need to be fine-tuned for every new generation of flash memory, making it difficult to design a system that universally meets performance requirements. Second, multiple FTL tasks generate sequences of flash memory requests that contend for the resources of the shared flash memory subsystem. This resource contention creates queueing delays that increase response times and causes long-tail latencies. Lastly, depending on the state of the flash storage, the importance of FTL tasks dynamically changes. For example, if the flash storage runs out of free blocks for writing host data, host request handling stalls and waits for garbage collection to reclaim free space. On the other hand, with sufficient free blocks, there is no incentive prioritizing garbage collection over host request handling.

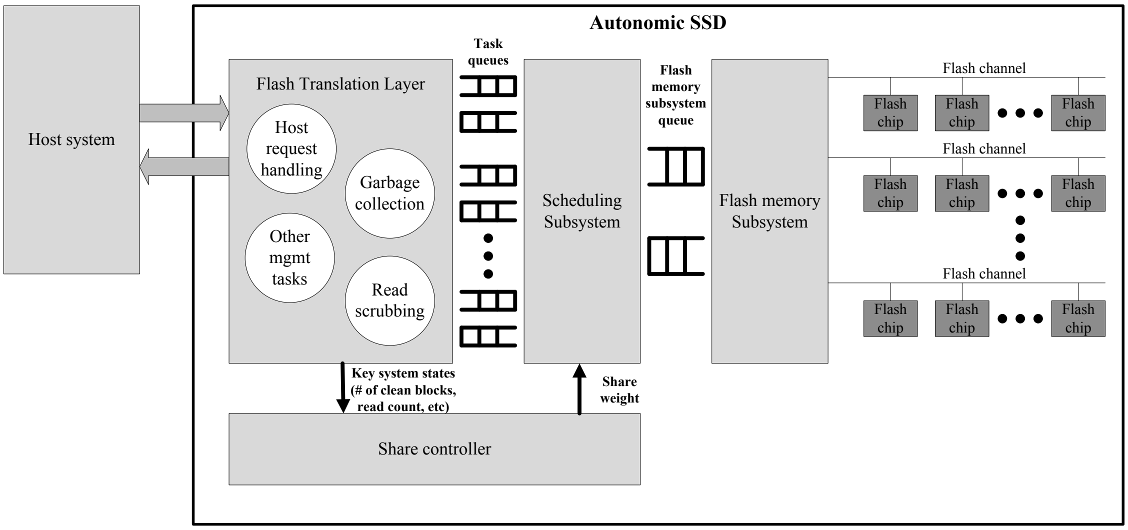


Figure 2: Overall architecture of AutoSSD and its components.

3 AutoSSD Architecture

In this section, we describe the overall architecture and design of the autonomic SSD (AutoSSD) as shown in Figure 3. In our model, all FTL tasks run concurrently, with each designed and implemented specifically for its job. Each task independently interfaces the scheduling subsystem, and the scheduler arbitrates the resources in the flash memory subsystem according to the assigned share. The share controller monitors key system states and determines the appropriate share for each FTL task. AutoSSD is agnostic to the specifics of the FTL algorithms (i.e., mapping scheme and GC victim selection), and the following subsections focus on the overall architecture and design that enable the self-management of the flash storage.

3.1 Virtualization of the Flash Memory Subsystem

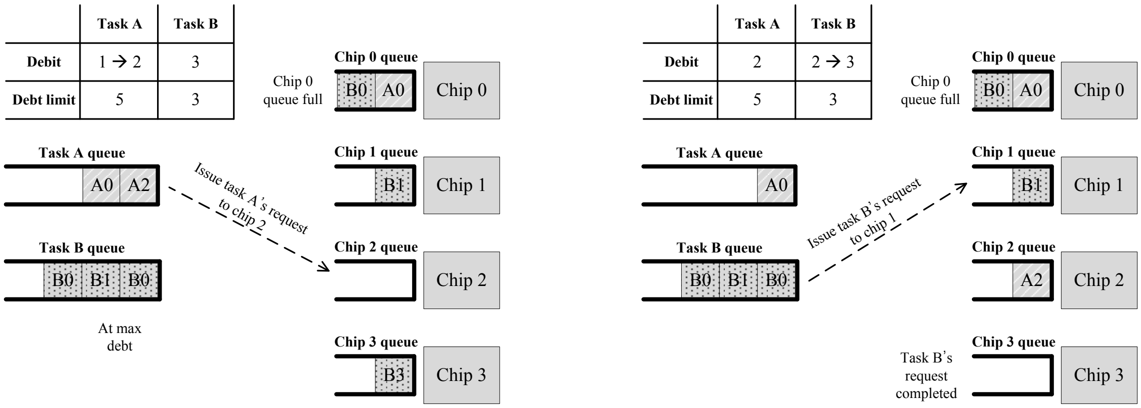
The architecture of AutoSSD allows each task to be independent of others by virtualizing the flash memory subsystem. Each FTL task is given a pair of request and response queues to send and receive flash memory requests and responses, respectively. This interface provides an illusion of a dedicated (yet slower) flash memory subsystem and allows an FTL task to generate flash memory requests oblivious of others (whether idle or active) or the requests they generate (intensity or which resources they are using). Details of the flash memory subsystem are completely abstracted by the scheduling subsystem, and only the back-pressure of the queue limits each task from generating more flash memory requests.

This virtualization not only effectively frees each task from having to worry about others, but also makes it easy to add a new FTL task to address any future flash memory quirks. While background operations such as garbage collection, read scrubbing, and wear leveling have similar flash memory workload patterns (reads and programs, and then erases), the objective of each task is distinctly different. Garbage collection reclaims space for writes, read scrubbing preventively relocates data to ensure data integrity, and wear leveling swaps contents of data to even out the damage done on flash memory cells. Our design allows seamless integration of new tasks without having to modify existing ones and reoptimize the system.

3.2 Scheduling for Share Enforcement

The scheduling subsystem interfaces with each FTL task and arbitrates the resources of the flash memory subsystem. The scheduler needs to be efficient with low overhead as it manages concurrency (tens and hundreds of flash memory requests) and parallelism (tens and hundreds of flash memory chips) at a small timescale.

In AutoSSD, we consider these unique domain characteristics and arbitrate the flash memory subsystem resource through *debit scheduling*. The debit scheduler tracks and limits the number of outstanding requests per task across all resources, and is based on the request windowing technique [14, 20, 34] from the disk scheduling domain. If the number of outstanding requests for a task, which we call debit, is under the limit, its requests are eligible to be issued; if it's not, the request cannot be issued until one or more of requests from that task completes. The debt limit is proportional to the share set by the share controller, allowing a task with a higher share to potentially utilize more resources simultaneously. The sum of all tasks' debt limit represents the total amount of



	Task A	Task B
Debit	1 → 2	3
Debt limit	5	3

	Task A	Task B
Debit	2	2 → 3
Debt limit	5	3

(a) Task A's request issued to Chip 2.

(b) Task B's request issued to Chip 0.

Figure 3: Two debit scheduling examples. In the scenario of Figure 3a, no more requests can be sent to Chip 0, and Task B is at its maximum debit. The only eligible scheduling is issuing Task A's request to Chip 2. In the scenario of Figure 3b, while Task A is still under the debt limit, its request cannot be issued to a chip with a full queue. On the other hand, a request from Task B can be issued as Chip 3's operation for Task B completes.

parallelism, and is set to the total number of requests that can be queued in the flash memory controller.

Figure 3 illustrates two scenarios of the debit scheduling. In both scenarios, the debt limit is set to 5 requests for Task A, and 3 for Task B. In Figure 3a, no more requests can be sent to Chip 0 as its queue is full, and Task B's requests cannot be scheduled as it is at its debt limit. Under this circumstance, Task A's request to Chip 2 is scheduled, increasing its debit value from 1 to 2. In Figure 3b, the active operation at Chip 3 for Task B completes, allowing Task B's request to be scheduled. Though Task B's request to Chip 1 is not at the head of the queue, it is scheduled out-of-order as there is no dependence between the requests to Chip 0 and Chip 1. Task A, although below the debt limit, cannot have its requests issued until Chip 0 finishes a queued operation, or until a new request to another chip arrives. Though not illustrated in these scenarios, when multiple tasks under the limit compete for the same resource, one is chosen with skewed randomness favoring a task with a smaller debit to debt limit ratio. Randomness is added to probabilistically avoid starvation.

Debit scheduling only tracks the number of outstanding requests, yet exhibits interesting properties. First, it can make scheduling decisions without complex computations and bookkeeping. This allows the debit scheduler to scale with increasing number of resources. Second, although it does not explicitly track *time*, it implicitly favors short latency operations as they have a faster turn-around-time. In scheduling disciplines such as weighted round robin [26] and weighted fair queueing (WFQ) [10], the latency of operations must be known or estimated to achieve some degree of fairness. Debit scheduling, however, approximates fairness in the time-domain only by tracking the number of outstanding requests. Lastly, the scheduler is in fact not work-conserving. The total debt limit can be scaled up to approximate a work-conserving scheduler, but the share-based resource reservation of the debit scheduler allows high responsiveness, as observed in the resource reservation protocol for Ozone [36].

3.3 Feedback Control of Share

The share controller determines the appropriate share for the scheduling subsystem by observing key system states. States such as the number of free blocks and the maximum read count reflect the stability of the flash storage. This is critical for the overall performance and reliability of the system, as failure to keep these states at a stable level can lead to an unbounded increase in response time or even data loss.

For example, if the flash storage runs out of free blocks, not only do host writes block, but also all other tasks that use flash memory programs stall: activities such as making mapping data durable and writing periodic checkpoints also depend on the garbage collection to reclaim free space. Even worse, a poorly constructed FTL may become deadlocked if GC is unable to obtain a free block to write the valid data from its victim. On the other hand, if a read count for a block exceeds its recommended limit, accumulated read disturbances can lead to data loss if the number of errors is beyond the error correction capabilities. In order to prevent falling into these adverse system conditions, the share controller monitors the system states and adjusts shares to control the rate of progress for individual FTL tasks, so that the system is maintained within stable levels.

AutoSSD uses feedback to adaptively determine the shares for the internal FTL tasks. While the values of key system states must be maintained at an adequate level, the shares of internal tasks must not be set too high such that they severely degrade the host performance. Once a task becomes active, it initially is allocated a small share. If this fails to maintain the current level of the system state, the share is gradually increased to counteract the momentum. The following control function is used to achieve this behavior:

$$S_A[t] = P_A \cdot e_A[t] + I_A \cdot S_A[t-1] \tag{1}$$

Where $S_A[t]$ is the share for task A at time t, $S_A[t-1]$ is the previous share for task A, P_A and I_A are two non-negative coefficients for task A, and $e_A[t]$ is the error value for task A at time t. The error value function for GC is defined as follows:

$$e_{GC}[t] = \max(0, target_{freeblk} - num_{freeblk}[t]) \tag{2}$$

With $target_{freeblk}$ set to the GC activation threshold, the share for GC S_{GC} starts out small. If the number of free blocks $num_{freeblk}[t]$ falls far below $target_{freeblk}$, the error function $e_{GC}[t]$ augmented by P_{GC} ramps up the GC share S_{GC}. After the number of free blocks $num_{freeblk}[t]$ exceeds the threshold $target_{freeblk}$, the share S_{GC} slowly decays given $I_{GC} < 1$.

Addition to the GC share control, the error value function for read scrubbing (RS) is defined as follows:

$$e_{RS}[t] = \max(0, \max_{i \in blk}(readcnt_i[t]) - target_{readcnt}) \tag{3}$$

Where $\max\limits_{i \in blk}(readcnt_i[t])$ is the maximum read count across all blocks in the system at time t, and $target_{readcnt}$ is the RS activation threshold.

In our design, the share for internal management schemes starts out small, anticipating host request arrivals and using the minimum amount of resources to perform its task. If the system state does not recover, the error (the difference between the desired and the actual system state values) accumulates, increasing the share over time.

It is important to note that the progress rate for a task depends not only on the share, but also on the workload, algorithm, and system state. For example, the number of valid data in the victim block, the location of the mapping data associated with the valid data, and the access patterns at the flash memory subsystem all affect the rate of progress for GC. A task's progress rate is, in fact, non-linear to the share under real-world workloads, and computationally solving for the optimal share involves large overhead, if not impossible. As a result, the two coefficients P and I for FTL tasks are empirically hand-tuned in this work.

Table 1: System configuration.

Parameter	Value	Parameter	Value
# of channels	4	Read latency	50μs
# of chips/channel	4	Program latency	500μs
# of planes/chip	2	Erase latency	5ms
# of blocks/plane	1024	Data transfer rate	400MB/s
# of pages/block	512	Physical capacity	256GB
Page size	16KB	Logical capacity	200GB

4 Evaluation Methodology and Modeling

We model a flash storage system on top of the DiskSim environment [1] by enhancing its SSD extension [3]. In this section, we describe the various components and configuration of the SSD, and the workload and test settings used for the evaluation.

4.1 Flash Memory Subsystem

Flash memory controller is based on Ozone [36] that fully utilizes flash memory subsystem's channel and chip parallelism. There can be at most four requests queued to each chip in the controller. Increasing this queue depth does not significantly increase intra-chip parallelism, as cached operations of flash memory have diminishing benefits as the channel bandwidth increases. Instead, a smaller queue depth is chosen to increase the responsiveness of the system.

Table 1 summarizes the default flash storage configuration used in our experiments. Of the 256GB of physical space, 200GB is addressable by the host system, giving an over-provisioning factor of 28%.

4.2 Flash Translation Layer

We implement core FTL tasks and features that are essential for storage functions, yet cause performance variations. Garbage collection reclaims space, but it degrades the performance of the system under host random writes. Read scrubbing that preventively relocates data creates background traffic on read-dominant workloads. Mapping table lookup is necessary to locate host data, but it increases response time on map cache misses.

Mapping. We implement an FTL with map caching [15] and a mapping granularity of 4KB. The entire mapping table is backed in flash, and mapping data, also maintained at the 4KB granularity, is selectively read into memory and written out to flash during runtime. The LRU policy is used to evict mapping data, and if the victim contains any dirty mapping entries, the 4KB mapping data is written to flash. By default, we use 128MB of memory to cache the mapping table. The second-level mapping that tracks the locations of the 4KB mapping

Table 2: Trace workload characteristics.

Workload	Duration (hrs)	Number of I/Os (Millions)		Average request size(KB)		Inter-arrival time (ms)	
		Write	Read	Write	Read	Average	Median
DAP-DS	23.5	0.1	1.3	7.2	31.5	56.9	31.6
DAP-PS	23.5	0.5	0.6	96.7	62.1	79.9	1.7
DTRS	23.0	5.8	12.0	31.9	21.8	4.6	1.5
LM-TBE	23.0	9.2	34.7	61.9	53.2	1.9	0.8
MSN-CFS	5.9	1.1	3.2	12.9	8.9	4.9	2.0
MSN-BEFS	5.9	9.2	18.9	11.6	10.7	0.8	0.3
RAD-AS	15.3	2.0	0.2	9.9	11.0	24.9	0.8
RAD-BE	17.0	4.3	1.0	13.0	106.2	11.7	2.6

data is always kept in memory as it is accessed more frequently and orders of magnitude smaller than the first-level mapping table.

Host request handling. Upon receiving a request, the host request handler looks up the second-level mapping to locate the mapping data that translates the host logical address to the flash memory physical address. If the mapping data is present in memory (hit), the host request handler references the mapping data and generates flash memory requests to service the host request. If it is a miss, a flash memory read request to fetch the mapping data is generated, and the host request waits until the mapping data is fetched. Host requests are processed in a non-blocking manner; if a request is waiting for the mapping data, other requests may be serviced out-of-order. In our model, if the host write request is smaller than the physical flash memory page size, multiple host writes are aggregated to fill the page to improve storage space utilization. We also take into consideration of the mapping table access overhead and processing delays. Mapping table lookup delay is set to be uniformly distributed between 0.5μs and 1μs, and the flash memory request generation delay for the host task is between 1μs and 2μs.

Garbage collection. The garbage collection (GC) task runs concurrently and independently from the host request handler and generates its own flash memory requests. Victim blocks are selected based on cost-benefit [40]. Once a victim block is selected, valid pages are read and programmed to a new location. Mapping data is updated as valid data is copied, and this process may generate additional requests (both reads and programs) for mapping management. Once all the valid pages have been successfully copied, the old block is erased and marked free. GC becomes active when the number of free blocks drops below a threshold, and stops once the number of free blocks exceeds another threshold, similar to the segment cleaning policy used for the log-structured file system [40]. In our experiments, the two threshold values for GC activation and deactivation are set to 128 and 256 free blocks, respectively. The garbage collection task also has a request generation delay, set to be uniformly distributed between 1μs and 3μs.

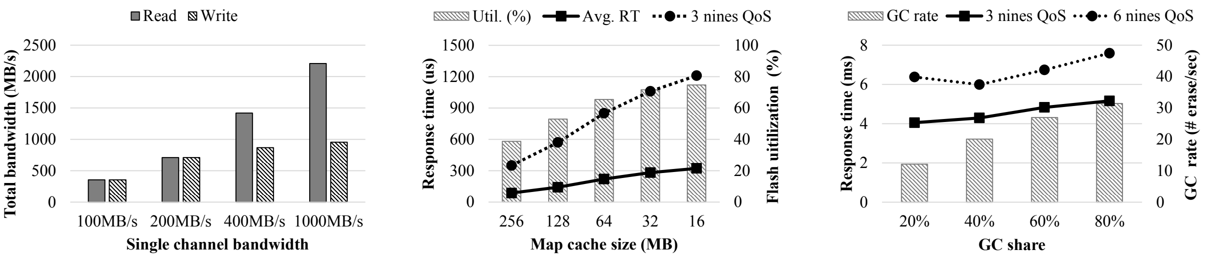
Read scrubbing. The read scrubbing (RS) task also runs as its own stand-alone task. Victims are selected greedily based on the read count of a block: the block with the most number of reads is chosen. Other than that, the process of copying valid data is identical to that of the garbage collection task. RS becomes active when the maximum read count of the system goes beyond a threshold, and stops once it falls below another threshold. The default threshold values for the activation and deactivation are set to 100,000 and 80,000 reads, respectively. Like the garbage collection task, the request generation delay (modeling the processing overhead of read scrubbing) is uniformly distributed between 1μs and 3μs.

4.3 Workload and Test Settings

We use both synthetic workloads and real-world I/O traces from Microsoft production servers [27] to evaluate the autonomic SSD architecture. Synthetic workloads of 128KB sequential accesses, 4KB random reads, and 4KB random read/writes are used to verify that our model behaves expectedly according to the system parameters.

From the original traces, the logical address of each host request is modified to fit into the 200GB range, and all the accesses are aligned to 4KB boundaries. All the traces are run for their full duration, with some lasting up to 24 hours and replaying up to 44 million I/Os. The trace workload characteristics are summarized in Table 2.

Prior to each experiment, the entire physical space is randomly written to emulate a pre-conditioned state so that the storage would fall under the steady state performance described in SNIA's SSS-PTS [2]. Furthermore, each block's read count is initialized with a non-negative random value less than the read scrubbing threshold to emulate past read activities.



(a) 128KB sequential accesses.

(b) 4KB random reads.

(c) 4KB random reads/writes.

Figure 4: Performance under synthetic workloads. Figure 4a shows the total bandwidth under 128KB sequential accesses with respect to changes in the channel bandwidth. Figure 4b shows the performance (average response time and 3 nines QoS) and the utilization of the flash memory subsystem with respect to changes in the size of the in-memory map cache. Figure 4c shows the performance (3 nines and 6 nines QoS) and the GC progress rate with respect to the GC share.

5 Experiment Results

This section presents experimental results under the configuration and workload settings described in the previous section. The main performance metric we report is the system response time seen at the I/O device driver. We first validate our SSD model using synthetic workloads, and then present experimental results with I/O traces. We replayed the I/O traces with the original request dispatch times, and with the dispatch times scaled down to increase the workload intensity.

5.1 Micro-benchmark Results

Figure 4 illustrates the performance of the autonomic SSD architecture (AutoSSD) with debit scheduling under four micro-benchmarks. Figure 4a plots the total bandwidth under 128KB sequential reads and 128KB sequential writes as we increase the channel bandwidth. As the channel bandwidth increases, the flash memory operation latency becomes the performance bottleneck. Write performance saturates early as the program latency cannot be hidden with data transfers. At 1000MB/s channel bandwidth, the read operation latency also becomes the bottleneck, unable to further extract bandwidth from the configured four channels. Traffic from GC and mapping management has a negligible effect for large sequential accesses, and RS task was disabled for this run to measure maximum raw bandwidth.

In Figure 4b, we vary the in-memory map cache size and measure the response times of 4KB random read requests when issued at 100K I/Os per second (IOPS). As expected, the response time is the smallest when the entire map is in memory, as it does not generate map read requests once the cache is filled after cold-start misses. However, as the map cache becomes smaller, the response time for host reads increases not only because it probabilistically stalls waiting for map reads from flash memory, but also due to increased flash memory traffic, which causes larger queueing delays.

Lastly, we demonstrate that the debit scheduling mechanism exerts control over FTL tasks in Figure 4c. In this scenario, 4KB random read/write requests are issued at 20K IOPS with a 1-to-9 read/write ratio. Both the response time of host read requests and GC task's progress (in terms of the number of erases per second while active) are measured at fixed GC shares from 20% to 80%. As shown by the bar graph, more blocks are erased as the share for GC increases. Furthermore, with more GC share, the overall host performance suffers, as evident by the increase in the 3 nines QoS. Deceptively, however, assigning not enough share to GC will result in worse tail latency as shown by the 6 nines QoS. GC needs to produce sufficient number of free blocks for the host to consume, and failure to do so will cause the host to block.

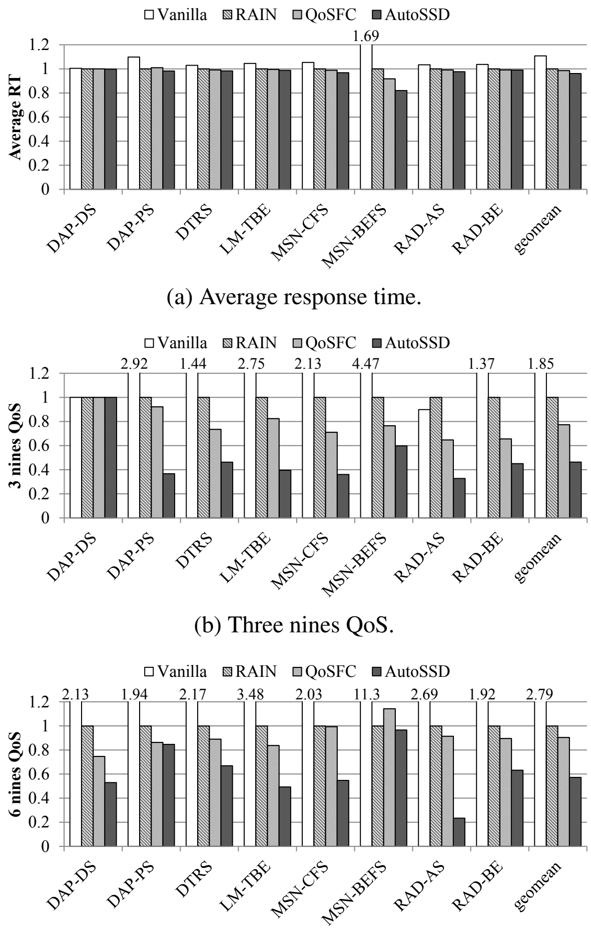
5.2 I/O Trace Results

Using I/O traces, we evaluate the performance of AutoSSD and compare it to following three systems:

Vanilla represents a design without virtualization and coordination, and all tasks dispatch requests to the controller through a single pair of request/response queue.

RAIN [45] uses parity to reconstruct data when accessing the chip is blocked by background tasks. Resources are arbitrated through fixed priority scheduling, with host requests having the highest priority. This technique requires an additional physical capacity to store parity data.

QoSFC [28] schedules using weighted fair queueing (WFQ) and represents a work-conserving system that does not reserve resources. It maintains virtual time as a measure of progress for each FTL task at each flash memory resource.



(a) Average response time.

(b) Three nines QoS.

(c) Six nines QoS.

Figure 5: Comparison of Vanilla, RAIN, QoSFC, and AutoSSD under eight different traces. Results are normalized to the performance of RAIN. AutoSSD reduces the average response time by up to 18.0% under MSN-BEFS (by 3.8% on average), the 3 nines QoS by up to 67.2% under RAD-AS (by 53.6% on average). and the 6 nines QoS by up to 76.6% under RAD-AS (by 42.7% on average).

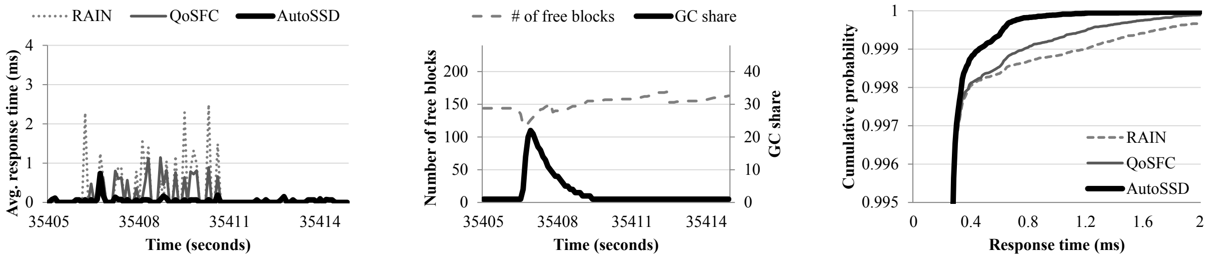
As the focus of this paper is response time characteristics, we only measure the performance of QoS-sensitive small reads (no larger than 64KB) in terms of the average response time, the 3 nines (99.9%) QoS figure, and the 6 nines (99.9999%).

Figure 5 compares the performance of the four systems under eight different traces. Compared to RAIN, AutoSSD reduces the average response time by up to 18.0% under MSN-BEFS as shown in Figure 5a. For the 3 nines QoS, AutoSSD shows improvements across most workloads, reducing it by 53.6% on average and as much as by 67.2% under RAD-AS (see Figure 5b). For the 6 nines QoS, AutoSSD shows much greater improvements, reducing it as much as by 76.6% under RAD-AS (see Figure 5c). Without coordination among FTL tasks, the Vanilla performance suffers, especially for the long tail latencies. In terms of the 6 nines, AutoSSD performs well under bursty workloads such as RAD-AS and LM-TBE (large difference between average and median inter-arrival times in Table 2). This is because AutoSSD limits the progress of internal FTL tasks depending on the state of the system, making resources available for the host in a non-work-conserving manner. This is in contrast to the scheduling disciplines used by the other systems: Vanilla uses FIFO scheduling; RAIN, priority scheduling; and QoSFC, weighted fair queueing.

To better understand the overall results in Figure 5, we microscopically examine the performance under RAD-AS in Figure 6 and LM-TBE in Figure 7. Figure 6a shows the average response time of three systems—RAIN, QoSFC, and AutoSSD—during a 10-second window, approximately 10 hours into RAD-AS. GC is active during this window for all the three systems, and both RAIN and QoSFC exhibit large spikes in response time. On the other hand, AutoSSD is better able to bound the performance degradation caused by an active garbage collection. Figure 6b shows the number of free blocks and the GC share during that window for AutoSSD. The sawtooth behavior for the number of free blocks is due to host requests consuming blocks, and GC gradually reclaiming space. GC share is reactively increased when the number of free blocks becomes low, thereby increasing the rate at which GC produces free blocks. If the number of free blocks exceeds the GC activation threshold, the share decays gradually to allow other tasks to use more resources. In effect, AutoSSD improves the overall response time as shown in Figure 6c.

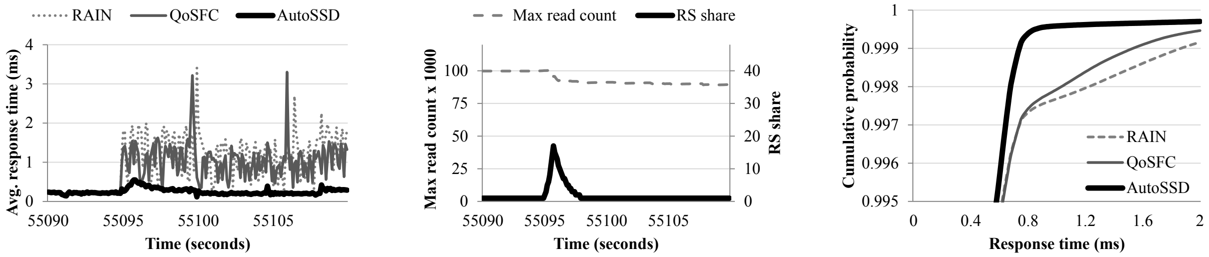
For LM-TBE, Figure 7a shows the average response time of the three systems during a 20-second window, approximately 15 hours into the workload. Here we observe read scrubbing (RS) becoming active due to the read-dominant characteristics of LM-TBE. We observe that both RAIN and QoSFC show large spikes in response time that lasts longer than the perturbation caused by GC for RAD-AS (cf. Figure 6a). While GC is incentivized to select a block with less valid data, RS is likely to pick a block with a lot of valid data that are frequently read but not frequently updated: this causes the performance degradation induced by RS to last longer than that by GC. AutoSSD limits this effect, while still decreasing the maximum read count in the system by dynamically adjusting the share of RS, as shown in Figure 7b. Figure 7c shows the response time CDF of the three systems.

Figure 8 illustrates the delay causes for the flash memory requests generated by the host request handling task under MSN-BEFS. Note that this is different from the response time of host requests: this shows the average wait time that a flash memory request (for servicing the host) experiences, broken down by different causes. Category



(a) Average response time under `RAD-AS`. (b) Change in GC share under `RAD-AS`. (c) Response time CDF under `RAD-AS`.

Figure 6: Comparison of RAIN, QoSFC, and AutoSSD under `RAD-AS`. Figure 6a shows the average response time sampled at 100ms in the selected 10-second window. Figure 6b shows the number of free blocks and the GC share of AutoSSD for the same 10-second window. Figure 6c plots the response time CDF for the entire duration.



(a) Average response time under `LM-TBE`. (b) Change in RS share under `LM-TBE`. (c) Response time CDF under `LM-TBE`.

Figure 7: Comparison of RAIN, QoSFC, and AutoSSD under `LM-TBE`. Figure 7a shows the average response time sampled at 100ms in the selected 20-second window. Figure 7b shows the maximum read count and the RS share of AutoSSD for the same 20-second window. Figure 7c plots the response time CDF for the entire duration.

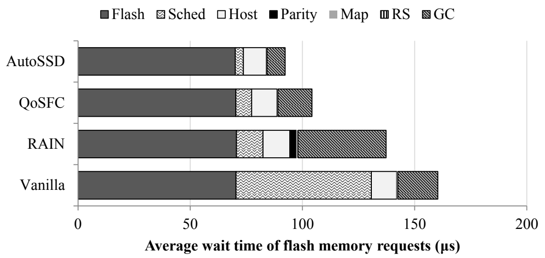


Figure 8: Breakdown of wait time experienced by flash memory requests under `MSN-BEFS`.

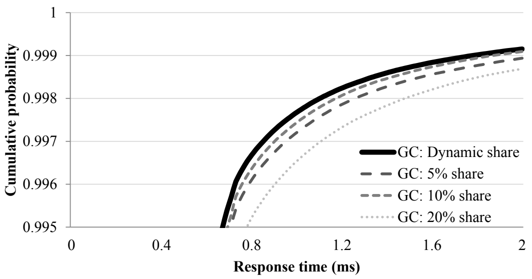
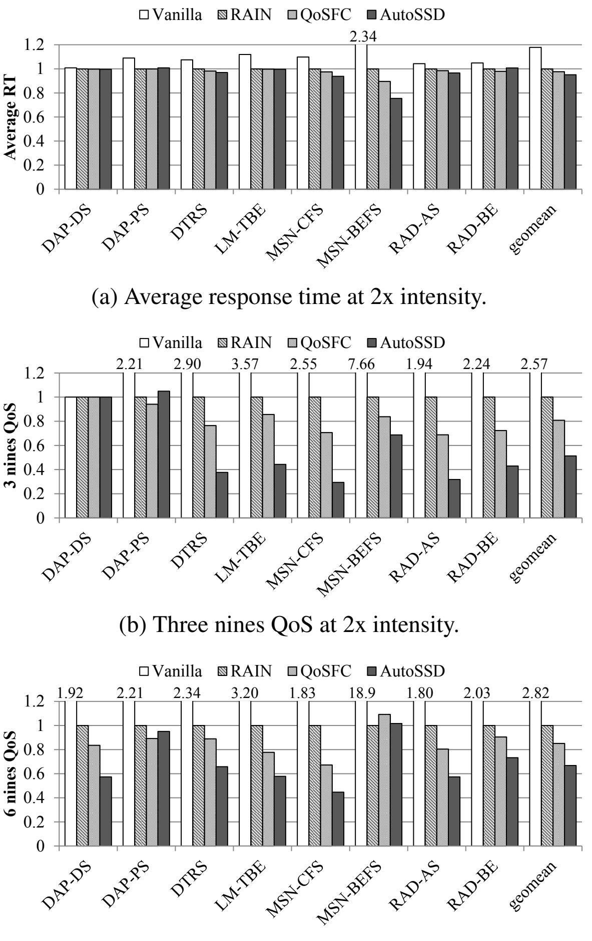


Figure 9: Comparison of AutoSSD with static shares and dynamic share under `MSN-BEFS`.

`Flash` represents flash memory latency, combining both flash array access latency and data transfers. `Sched` is the time spent waiting to be scheduled, either waiting in the queue because the target queue is full, or waiting because the scheduler limits the progress in a non-work-conserving manner (the case for AutoSSD). The large `Sched` wait time for Vanilla is caused by uncoordinated sharing of resources, while that for AutoSSD is small as the scheduler reserves resources for host requests. The remaining five categories are delays experienced due to resource blocking. Most noticeably, the wait time caused by GC in RAIN is higher than the other systems. When RAIN generates alternate flash memory requests to reconstruct data through parity, these additional requests can, in turn, be blocked again at another resource. In ttFlash [45], this problem is overcome by statically limiting the number of active GC to one per parity group. This technique is not used in our evaluation as a fixed cap on the number of allowed GC can quickly deplete free blocks, especially for high-intensity small random write workloads.

Next, we examine the effectiveness of the dynamic share assignment over the static ones. Figure 9 shows the response time CDF of AutoSSD under `MSN-BEFS` with static shares of 5%, 10%, and 20% for GC, along with the share controlled dynamically. As illustrated by the gray lines, decreasing the GC share from 20% to 10% improves the overall performance. However, when further reducing the GC share to 5%, we observe that the curve for 5% dwindles as it approaches higher QoS and performs worse than the 10% curve. This indicates that while a lower GC share achieves better performance at lower QoS levels, a higher GC share is desirable to reduce long-tail latencies as it generates free blocks at a higher rate, preventing the number of free blocks from becoming critically low. This observation is in accordance with the performance under synthetic workload in Figure 4c. Using feedback control to adjust the GC share dynamically shows better performance over all the static values, as it can adapt to an appropriate share value by reacting to the changes in the system state.



(a) Average response time at 2x intensity.

(b) Three nines QoS at 2x intensity.

(c) Six nines QoS at 2x intensity.

Figure 10: Comparison of Vanilla, RAIN, QoSFC, and AutoSSD under eight different traces at 2x intensity. Results are normalized to the performance of RAIN at 2x intensity. AutoSSD reduces the average response time by up to 24.6% under MSN-BEFS (by 4.9% on average), the 3 nines QoS by up to 70.6% under MSN-CFS (by 48.6% on average). and the 6 nines QoS by up to 55.3% under MSN-CFS (by 33.2% on average).

5.3 I/O Trace Results at Higher Intensity

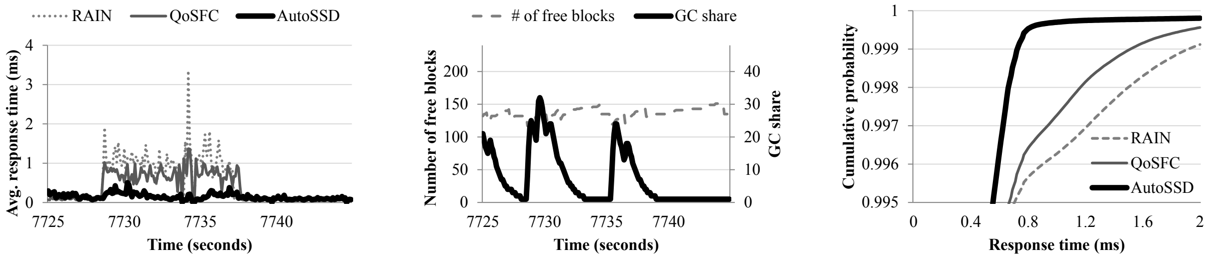
In this subsection, we present experimental results with higher request intensities. Here, the request dispatch times are reduced in half, but other parameters such as the access type and the target address remain unchanged. This experiment is intended to examine the performance of the four systems—Vanilla, RAIN, QoSFC, and AutoSSD—under a more stressful scenario.

Figure 10 compares the performance in the new setting. AutoSSD reduces the average response time by up to 24.6% under MSN-BEFS (see Figure 10a), the 3 nines QoS figure by 48.6% on average and as much as 70.6% under MSN-CFS (see Figure 10b), and the 6 nines QoS figure by as much as 55.3% under MSN-CFS (see Figure 10c). With workload intensity increased, the overall improvement in long tail latency decreases due to a smaller wiggle room for AutoSSD to manage FTL tasks. This is especially true for high-intensity workloads such as MSN-BEFS: with host requests arriving back-to-back (cf. halve the inter-arrival time in Table 2), debit scheduling has little advantage over other scheduling schemes. However, AutoSSD nevertheless outperforms prior techniques across the diverse set of workloads. Workloads such as RAD-AS and LM-TBE that showed the most reduction in long tail latency under the original intensity (cf. Figure 5c) still exhibit performance improvements with AutoSSD in the 6 nines, even with increased workload intensity.

We examine DTRS more closely in Figure 11. Figure 11a shows the average response time of the three systems—RAIN, QoSFC, and AutoSSD—during a 20-second window, approximately 2 hours into the workload. GC is active during this window for all the three systems, and AutoSSD is better able to bound the performance degradation caused by an active garbage collection, while both RAIN and QoSFC exhibit large spikes in response time. Figure 11b shows the number of free blocks and the GC share during that window for AutoSSD. Similar to the results in the previous section, the share for GC reactively increases at a lower number of blocks, and decays once the number of free blocks reaches a stable region. Again, the number of free blocks shows a sawtooth behavior, and the ridges of GC share curve matches the valleys of the free block curve. Figure 11c plots the response time CDF of the three systems, demonstrating the effectiveness of our dynamic management.

6 Discussion and Related Work

There are several studies on real-time performance guarantees of flash storage, but they depend on RTOS support [7], specific mapping schemes [8, 39, 46], a number of reserve blocks [8, 39], and flash operation latencies [46]. These tight couplings make it difficult to extend performance guarantees when system requirements



(a) Average response time under 2xDTRS. (b) Change in GC share under 2xDTRS. (c) Response time CDF under 2xDTRS.

Figure 11: Comparison of the RAIN, QoSFC, and AutoSSD under DTRS running at 2x intensity. Figure 11a shows the average response time sampled at 100ms in the selected 20-second window. Figure 11b shows the number of free blocks and the GC share of AutoSSD for the same 20-second window. Figure 11c plots the response time CDF for the entire duration.

and flash memory technology change. On the other hand, our architecture is FTL implementation-agnostic, allowing it to be used across a wide range of flash devices and applications.

Some techniques focus on when to perform GC (based on threshold [31], slack [22], or host idleness [25, 37]). These approaches complement our design that focuses on the fine-grained scheduling and dynamic management of multiple FTL tasks running concurrently. By incorporating workload prediction techniques to our design, we can extend AutoSSD to increase the share on background tasks when host idleness is expected, and decrease it when host requests are anticipated.

Exploiting redundancy to reduce performance variation has been studied in a number of prior art. Harmonia [30] and Storage engine [42] duplicate data across multiples SSDs, placing one in read mode and the other in write mode to eliminate GC's impact on read performance. ttFlash [45] uses multiple flash memory chips to reconstruct data through a RAID-like parity scheme. Relying on redundancy effectively reduces the storage utilization, but otherwise complements our design of dynamic management of various FTL tasks.

Performance isolation aims to reduce performance variation caused by multiple hosts through partitioning resources (vFlash [43], FlashBlox [18]), improving GC efficiency by grouping data from the same source (Multistreamed [24], OPS isolation [29]), and penalizing noisy neighbors (WA-BC [21]). These performance isolation techniques are complementary to our approach of fine-grained scheduling and dynamic management of concurrent FTL tasks.

The design of the autonomic SSD architecture borrows ideas from prior work on shared disk-based storage systems such as Façade [33], PARDA [14], and Maestro [34]. These systems aim to meet performance requirements of multiple clients by throttling request rates and dynamically adjusting the bound through a feedback control. However, while these disk-based systems deal with fair sharing of disk resources among multiple hosts, we address the interplay between the foreground (host I/O) and the background work (garbage collection and other management schemes).

Aqueduct [32] and Duet [4] address the performance impact of background tasks such as backup and data migration in disk-based storage systems. However, background tasks in flash storage are triggered at a much smaller timescale, and SSDs uniquely create scenarios where the foreground task depends on the background task, necessitating a different approach.

7 Conclusion

In this paper, we presented the design of an autonomic SSD architecture that self-manages concurrent FTL tasks in the flash storage. By judiciously coordinating the use of resources in the flash memory subsystem, the autonomic SSD manages the progress of concurrent FTL tasks and maintains the internal system states of the storage at a stable level. This self-management prevents the SSD from falling into a critical condition that causes long tail latency. In effect, AutoSSD reduces the average response time by up to 18.0%, the 3 nines (99.9%) QoS by up to 67.2%, and the 6 nines (99.9999%) QoS by up to 76.6% for QoS-sensitive small reads.

Acknowledgment

We thank the anonymous reviewers for their constructive and insightful comments, and also thank Jaejin Lee, Jongmoo Choi, Hyeonsang Eom, and Eunji Lee for reviewing the early stages of this work. This work was supported in part by SK Hynix and the National Research Foundation of Korea under the PF Class Heterogeneous High Performance Computer Development (NRF-2016M3C4A7952587). Institute of Computer Technology at Seoul National University provided the research facilities for this study.

References

[1] The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101). `http://www.pdl.cmu.edu/PDL-FTP/DriveChar/CMU-PDL-08-101.pdf`, 2008. Parallel Data Laboratory.

[2] Solid state storage (SSS) performance test specification (PTS) enterprise. `http://snia.org/sites/default/files/SSS_PTS_Enterprise_v1.1.pdf`, 2013. Storage Networking Industry Association.

[3] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M. S., and Panigrahy, R. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference* (2008), pp. 57–70.

[4] Amvrosiadis, G., Brown, A. D., and Goel, A. Opportunistic storage maintenance. In *ACM Symposium on Operating Systems Principles* (2015), pp. 457–473.

[5] Aritome, S. *NAND flash memory technologies*. John Wiley & Sons, 2015.

[6] Cai, Y., Haratsch, E. F., Mutlu, O., and Mai, K. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Design, Automation and Test in Europe* (2012), pp. 521–526.

[7] Chang, L.-P., Kuo, T.-W., and Lo, S.-W. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems 3*, 4 (2004), 837–863.

[8] Choudhuri, S., and Givargis, T. Deterministic service guarantees for NAND flash using partial block cleaning. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (2008), pp. 19–24.

[9] Dean, J., and Barroso, L. A. The tail at scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[10] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM* (1989), pp. 1–12.

[11] Gal, E., and Toledo, S. Algorithms and data structures for flash memories. *ACM Computing Surveys 37*, 2 (2005), 138–163.

[12] Grupp, L. M., Caulfield, A. M., Coburn, J., Swanson, S., Yaakobi, E., Siegel, P. H., and Wolf, J. K. Characterizing flash memory: anomalies, observations, and applications. In *IEEE/ACM International Symposium on Microarchitecture* (2009), pp. 24–33.

[13] Grupp, L. M., Davis, J. D., and Swanson, S. The bleak future of NAND flash memory. In *USENIX Conference on File and Storage Technologies* (2012), pp. 17–24.

[14] Gulati, A., Ahmad, I., Waldspurger, C. A., et al. PARDA: Proportional allocation of resources for distributed storage access. In *USENIX Conference on File and Storage Technologies* (2009), pp. 85–98.

[15] Gupta, A., Kim, Y., and Urgaonkar, B. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 229–240.

[16] Ha, K., Jeong, J., and Kim, J. An integrated approach for managing read disturbs in high-density NAND flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35*, 7 (2016), 1079–1091.

[17] Hao, M., Soundararajan, G., Kenchammana-Hosekote, D. R., Chien, A. A., and Gunawi, H. S. The tail at store: A revelation from millions of hours of disk and SSD deployments. In *USENIX Conference on File and Storage Technologies* (2016), pp. 263–276.

[18] Huang, J., Badam, A., Caulfield, L., Nath, S., Sengupta, S., Sharma, B., and Qureshi, M. K. FlashBlox: Achieving both performance isolation and uniform lifetime for virtualized SSDs. In *USENIX Conference on File and Storage Technologies* (2017), pp. 375–390.

[19] Jimenez, X., Novo, D., and Ienne, P. Wear unleveling: improving NAND flash lifetime by balancing page endurance. In *USENIX Conference on File and Storage Technologies* (2014), pp. 47–59.

[20] Jin, W., Chase, J. S., and Kaur, J. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS* (2004), pp. 37–48.

[21] Jun, B., and Shin, D. Workload-aware budget compensation scheduling for NVMe solid state drives. In *IEEE Non-Volatile Memory System and Applications Symposium* (2015), pp. 1–6.

[22] Jung, M., Choi, W., Srikantaiah, S., Yoo, J., and Kandemir, M. T. HIOS: A host interface I/O scheduler for solid state disks. In *ACM International Symposium on Computer Architecture* (2014), pp. 289–300.

[23] Jung, M., and Kandemir, M. Revisiting widely held SSD expectations and rethinking system-level implications. In *ACM International Conference on Measurement and Modeling of Computer Systems* (2013), pp. 203–216.

[24] Kang, J.-U., Hyun, J., Maeng, H., and Cho, S. The multi-streamed solid-state drive. In *USENIX Workshop on Hot Topics in Storage and File Systems* (2014).

[25] Kang, W., Shin, D., and Yoo, S. Reinforcement learning-assisted garbage collection to mitigate long-tail latency in SSD. *ACM Transactions on Embedded Computing Systems 16*, 5s (2017), 134.

[26] Katevenis, M., Sidiropoulos, S., and Courcoubetis, C. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *IEEE Journal on selected Areas in Communications 9*, 8 (1991), 1265–1279.

[27] Kavalanekar, S., Worthington, B., Zhang, Q., and Sharda, V. Characterization of storage workload traces from production Windows servers. In *IEEE International Symposium on Workload Characterization* (2008), pp. 119–128.

[28] Kim, B. S., and Min, S. L. QoS-aware flash memory controller. In *IEEE Real-Time and Embedded Technology and Applications Symposium* (2017), pp. 51–62.

[29] Kim, J., Lee, D., and Noh, S. H. Towards SLO complying SSDs through OPS isolation. In *USENIX Conference on File and Storage Technologies* (2015), pp. 183–189.

[30] Kim, Y., Oral, S., Shipman, G. M., Lee, J., Dillow, D. A., and Wang, F. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *IEEE Symposium on Mass Storage Systems and Technologies* (2011), pp. 1–12.

[31] Lee, J., Kim, Y., Shipman, G. M., Oral, S., Wang, F., and Kim, J. A semi-preemptive garbage collector for solid state drives. In *IEEE International Symposium on Performance Analysis of Systems and Software* (2011), pp. 12–21.

[32] Lu, C., Alvarez, G. A., and Wilkes, J. Aqueduct: Online data migration with performance guarantees. In *USENIX Conference on File and Storage Technologies* (2002), pp. 219–230.

[33] Lumb, C. R., Merchant, A., and Alvarez, G. A. Façade: Virtual storage devices with performance guarantees. In *USENIX Conference on File and Storage Technologies* (2003), pp. 131–144.

[34] Merchant, A., Uysal, M., Padala, P., Zhu, X., Singhal, S., and Shin, K. Maestro: quality-of-service in large disk arrays. In *ACM International Conference on Autonomic Computing* (2011), pp. 245–254.

[35] MEZA, J., WU, Q., KUMAR, S., AND MUTLU, O. A large-scale study of flash memory failures in the field. In *ACM SIGMETRICS* (2015), pp. 177–190.

[36] NAM, E. H., KIM, B. S. J., EOM, H., AND MIN, S. L. Ozone (O3): An out-of-order flash memory controller architecture. *IEEE Transactions on Computers 60*, 5 (2011), 653–666.

[37] PARK, S.-H., KIM, D.-G., BANG, K., LEE, H.-J., YOO, S., AND CHUNG, E.-Y. An adaptive idle-time exploiting method for low latency NAND flash-based storage devices. *IEEE Transactions on Computers 63*, 5 (2014), 1085–1096.

[38] PRINCE, B. *Vertical 3D memory technologies*. John Wiley & Sons, 2014.

[39] QIN, Z., WANG, Y., LIU, D., AND SHAO, Z. Real-time flash translation layer for NAND flash memory storage systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium* (2012), pp. 35–44.

[40] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[41] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. Flash reliability in production: The expected and the unexpected. In *USENIX Conference on File and Storage Technologies* (2016), pp. 67–80.

[42] SHIN, W., KIM, M., KIM, K., AND YEOM, H. Y. Providing QoS through host controlled flash SSD garbage collection and multiple SSDs. In *International Conference on Big Data and Smart Computing* (2015), pp. 111–117.

[43] SONG, X., YANG, J., AND CHEN, H. Architecting flash-based solid-state drive for high-performance I/O virtualization. *IEEE Computer Architecture Letters 13*, 2 (2014), 61–64.

[44] TSENG, H.-W., GRUPP, L., AND SWANSON, S. Understanding the impact of power loss on flash memory. In *Design Automation Conference* (2011), pp. 35–40.

[45] YAN, S., LI, H., HAO, M., TONG, M. H., SUNDARARAMAN, S., CHIEN, A. A., AND GUNAWI, H. S. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *USENIX Conference on File and Storage Technologies* (2017), pp. 15–28.

[46] ZHANG, Q., LI, X., WANG, L., ZHANG, T., WANG, Y., AND SHAO, Z. Optimizing deterministic garbage collection in NAND flash storage systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium* (2015), pp. 14–23.

[47] ZHENG, M., TUCEK, J., QIN, F., AND LILLIBRIDGE, M. Understanding the robustness of SSDs under power fault. In *USENIX Conference on File and Storage Technologies* (2013), pp. 271–284.

Geriatrix: Aging what you see and what you don't see

A file system aging approach for modern storage systems

Saurabh Kadekodi[1] Vaishnavh Nagarajan[1] Gregory R. Ganger[1] Garth A. Gibson[1,2]
[1]Carnegie Mellon University [2]Vector Institute

Abstract

File system performance on modern primary storage devices (Flash-based SSDs) is greatly affected by aging of the free space, much more so than were mechanical disk drives. We introduce *Geriatrix*, a simple-to-use profile driven file system aging tool that induces target levels of fragmentation in both allocated files (what you see) and remaining free space (what you don't see), unlike previous approaches that focus on just the former. This paper describes and evaluates the effectiveness of Geriatrix, showing that it recreates both fragmentation effects better than previous approaches. Using Geriatrix, we show that measurements presented in many recent file systems papers are higher than should be expected, by up to 30% on mechanical (HDD) and up to 80% on Flash (SSD) disks. Worse, in some cases, the performance rank ordering of file system designs being compared are different from the published results.

Geriatrix will be released as open source software with eight built-in aging profiles, in the hopes that it can address the need created by the increased performance impact of file system aging in modern SSD-based storage.

1 Introduction

The performance of a file system (FS) usually deteriorates over time. As FSs experience heavy churn, techniques such as write-back caching to expedite writes [38, 31, 14], data prefetching to assist reads [8, 35] and self-balancing data structures to contain search times [10] may pay for faster normal path performance now with more complex and fragmented on-device images as the system ages. An important factor affecting aged FS performance is poor FS layout [43, 42].

Naturally, therefore, FS benchmarking should consider the effects of aging. But, despite it being an important issue known for over twenty years [44], most research and benchmarks still ignore aging. For example, 65% (13 of 20) recent FS papers we examined (Table 1) neither mention aging nor include it in their evaluations. Unsurprisingly, our experiments confirm that aging continues to be a critical factor for FS performance.

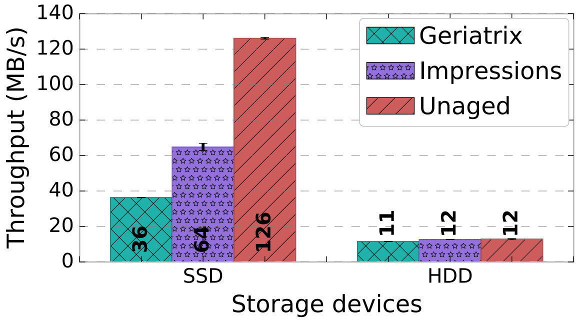


Figure 1: Aging impact on Ext4 atop SSD and HDD. The three bars for each device represent the FS freshly formatted (unaged), aged with Geriatrix, and aged with Impressions [2]. Although relatively small differences are seen with the HDD, aging has a big impact on FS performance on the SSD. Although their file fragmentation levels are similar, the higher free space fragmentation produced by Geriatrix induces larger throughput reductions than for Impressions. The experimental setup is detailed in Section 6.

While aging's overall importance has not waned, the particular aspects that have the most impact have changed over time, making previous aging approaches... stale. Previous general purpose aging approaches, from the work of Smith and Seltzer [44] to the state-of-the-art *Impressions* [2] tool, focus on achieving representative levels of file fragmentation. Such fragmentation exists when sequential blocks of a file or related metadata / files are scattered among logical block addresses (LBAs) of underlying storage. For FSs atop HDDs, with time-consuming mechanical positioning costs for accessing scattered LBAs, these file fragmentation effects are of most concern. For the Flash-based SSDs that now dominate primary and performance-tier deployments, these effects are less significant, due to LBA remapping and absence of mechanical positioning.

This paper introduces *Geriatrix*, a new FS aging tool for modern storage. In addition to file fragmentation, Geriatrix aggressively induces free space fragmentation and thereby even ages any underlying device remapping structures. As a result, it can recreate the much more significant aging effects seen with SSDs in real systems [36]. As one example, Figure 1 compares three instances of an Ext4 FS: Unaged, aged by Impressions, and aged by Geriatrix. On the HDD, fairly minor performance differences are observed for this particular benchmark, since the workload involves small files and relatively little access locality. On the SSD, however, large differences can be seen, and the Geriatrix-aged FS produces much greater aging effects.

Additional evaluation and experiments, later in the paper, confirm that these greater Geriatrix-induced effects are consistent and correctly representative. In addition, we recreated experiments from recent papers, showing both, that the reported performance fails to represent realistic expectations **and** that the rank ordering of configurations compared is sometimes changed. Figure 2 shows one such re-evaluation, in which we observe both effects.

Geriatrix uses a sophisticated profile-driven approach that ages a FS according to a reference (old) FS. This paper describes how Geriatrix extracts information from a profile and exercises the FS to recreate its fragmentation properties. With both theoretical analysis and experimental comparison to real FS images used as profiles, we show that Geriatrix faithfully reproduces both file and free space fragmentation.

Geriatrix is being released as an open source tool, together with eight built-in aging profiles and a repository of aged images of popular FSs. We hope that its availability will help increase the use of aging in FS benchmarking.

This paper makes three primary contributions. First, it exposes the impact of free space fragmentation and device aging for FSs on SSDs and the failure of existing aging approaches to recreate them. Second, it describes a new aging approach, embodied in Geriatrix, and confirms that it does faithfully recreate these aging effects. Third, it provides extensive evidence, including recreating recently published comparisons and showing that results change, of why aging must be part of benchmarking and offers Geriatrix as an open-source tool for doing so.

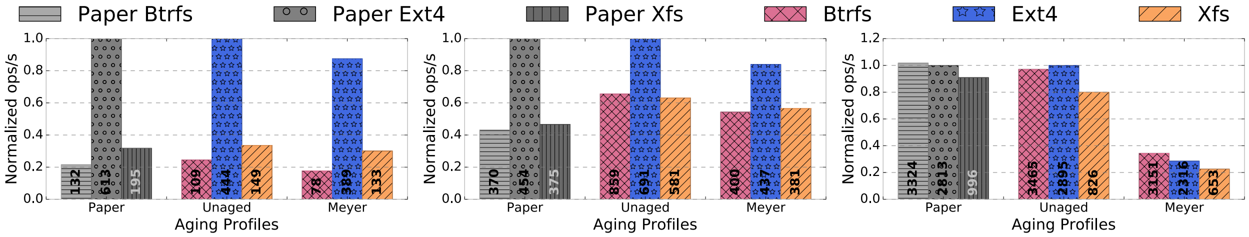
2 Related work

We classify aging tools into three categories: trace replay tools, scripts executing real-world applications and synthetic workload generators.

Trace replay tools are best used with FSs expecting a highly specialized workload. Traces can be captured and replayed at multiple levels - the network level [57], file level [32], FS level [40, 4], system call level [50], VFS level [20] and also at the block level [7]. Low level traces are typically FS specific resulting in loss of usefulness for comparing different FSs. Moreover, long traces are not widely available and are hard to capture. Trace replay tools rank high on reproducibility but do not represent all workloads.

The Andrew benchmark [17], Compilebench [27] and the Git-Benchmark [11, 12] are application benchmarks that implicitly involve some aging. These tools emulate user behavior by performing typical activities like extracting archives, reading files, compiling code, making directories, cloning repositories, etc. Compilebench performs these tasks on Linux kernel sources, while the Git-Benchmark can be run using any git repository. Tools in this category only exercise one workload pattern.

Geriatrix belongs to the category of synthetic workload generators, which also comprises of Smith and Seltzer's aging tool [44] and Impressions [2]. Smith's tool ages by recreating each file in a given reference snapshot and then performing creates and deletes according to the deltas observed in successive reference snapshots. It was one of the first tools to point out the degradation of FS performance with age. Impressions on the other hand is a realistic FS image creator that focuses on several FS characteristics including file size and directory depth distributions along with file attributes and contents. These tools take reference from already old FSs in order to perform aging.



(a) Btrfs HDD Varmail

(b) Btrfs HDD Webserver

(c) Btrfs SSD Fileserver

Figure 2: All three graphs reproduce experiments from the Btrfs ACM TOS publication [39] on aged FS instances. The *Paper* bars are normalized to the paper's Ext4 FS measurements, while the experiments we recreated are normalized to the unaged Ext4 performance on our hardware. Figures 2a and 2b show aging experiments performed on a HDD using the varmail and webserver profiles respectively. 2a shows modest slowdown (30% in Btrfs, 13% in Ext4 and 9% in Xfs), but preserves the rank ordering published in the paper. 2b disrupts published rank ordering and displays slowdowns of 17% in Btrfs and Ext4 and 12% in Xfs. Figure 2c shows effects of aging on SSD for the fileserver profile. Here too the published rank ordering is not preserved, but we observe massive slowdowns of 65% in Btrfs and 72% in Ext4, Xfs.

File System	Publication	Needed Aging?	Performed Aging?
yFS [55]	FAST 2003	Yes	Yes
Nilfs2 [21]	SIGOPS 2006	Yes	No
TFS [9]	FAST 2007	Yes	Yes
Data Domain Dedup FS [56]	FAST 2008	Yes	Yes
Panasas Parallel FS [51]	FAST 2008	Yes	Yes
CA-NFS [5]	FAST 2009	Yes	No
HYDRAStor [46]	FAST 2010	Yes	No
DFS [19]	FAST 2010	Yes	No
SFS [34]	FAST 2012	Yes	Yes
BlueSky [47]	FAST 2012	Maybe	No
ZZFS [29]	FAST 2012	Maybe	No
Nested FS in Virt. Env. [23]	FAST 2012	Yes	No
Btrfs [39]	ACM TOS 2013	Yes	No
ReconFS [26]	FAST 2014	Yes	No
F2fs [24]	FAST 2015	Yes	No
App. Managed Flash [25]	FAST 2016	Maybe	No
NOVA [52]	FAST 2016	Yes	No
CFFS [54]	FAST 2016	Maybe	Yes
BetrFS [11, 18, 53]	FAST 2015, 2016	Yes	Yes in [11]
Strata [22]	SOSP 2017	Yes	No

Table 1: A subset of major FS publications and whether their paper reports aging experiments. The *Needed Aging?* column is our understanding of whether aging could have affected published results, i.e. was aging (or commentary about it) necessary as a part of benchmarking. The *Performed Aging?* column refers to whether any aging-like experiment was performed. Our analysis reveals two FSs - yFS [55], TFS [9] performed long-running aging experiments; Data Domain FS [56] and Panasas FS [51] had production data (and therefore had seen aging in the field), SFS [34] ran a workload twice the size of the disk and CFFS [54] ran a large trace for aging. BetrFS [11] aged using the Git-benchmark (see § 3), which highlighted file fragmentation caused by age, but induced limited free space fragmentation. The remaining 13 papers do not discuss aging or its effects on their FSs.

3 Why do we need another aging tool?

Aging any software artifact implies understanding how it will stand the test of time. Aging is used for several reasons, such as uncovering performance deterioration with use, stressing the robustness of the software, and identifying fault tolerance and scalability issues. This section discusses four aspects of aging that current approaches insufficiently satisfy.

Free space fragmentation. State-of-the-art FS aging tools either replay a trace of FS commands or run scripts of important applications. Smith's aging tool [44] and Impressions [2] come close to what we expect from an aging tool. But, Smith's tool is a twenty year old artifact with dependencies on the Fast FS (FFS) [31]. Impressions matches an impressive number of aged metrics, but is focused on generating realistic FS content, not FS layout. These tools only target file fragmentation using a metric called *layout score*, that measures the disk contiguity of files after aging.

Git-benchmark [11, 12] is a recently published aging benchmark that ages the FS by cloning a git repository, repeatedly patching code files (via git pulls) and finally grepping for random strings in the patched repository. In this study as well, the authors only measure file fragmentation by extending the layout score (which they call the *dynamic layout score*) which additionally accounts for the contiguity in the FS access pattern for a file.

We ran the Git-benchmark from [12] on a 20GB Ext4 partition and observed a $> 7\times$ slowdown when grepping for the same string after 3000 git pulls versus grepping for a string after a single git pull on a fresh Ext4 partition. In order to understand the dramatic slowdowns this workload experiences, we traced the Ext4 kernel functions to find, where in the code, most of the time was spent during the arbitrary greps. Function tracing revealed that *ext4_es_lookup_extent* is the function where most of the time is spent during grep, i.e. looking up a FS data structure. Since the capacity utilization at the end of 3000 git pulls was only 4%, we measured the free space fragmentation and observed that $> 75\%$ of the free space extents were between 1-2GB. Thus, although the Git-benchmark successfully caused some file fragmentation, it caused little of the free space fragmentation prevalent in aged FSs.

To the best of our knowledge, the above mentioned tools are the only general purpose FS aging tools; and none of them produce the required free space fragmentation which is an inevitable consequence of aging. Geriatrix's fills this void by inducing adequate free space fragmentation, proof of which is shown in § 5 and whose effect in aging SSDs has been exemplified in Figure 1.

Device aging. SSDs contain a complex translation layer in the device firmware called a flash translation layer (FTL). FTLs are the reason that an SSD can act as a drop-in replacement for an HDD despite having entirely different hardware. FTLs primarily perform the tasks of address remapping, garbage collection and wear leveling. SSD device characteristics force the FTL to operate very similarly to a complex log-structured FS [41]. With time, the increased garbage collection work (interfering with the foreground work) combined with fragmented FTL mapping tables can hurt performance. Thus, in the case of an SSD, two systems are aging simultaneously, the FTL and the FS that the SSD has been formatted with. Since FTLs are proprietary, users typically have no insight into how well (or poorly) an FTL has aged.

Shingled magnetic recording (SMR) is a new hard drive architecture wherein adjacent tracks on a HDD are partially overlapped to increase the number of tracks on the disk, thus increasing the disk capacity. This technology was an answer to the traditional HDDs having surpassed the superparamagnetic effect [45], which disallows increasing sectors-per-track in order to achieve larger disk capacities. Commercially available SMR HDDs have a firmware different from, but as complicated as the FTL. Aghayev et al. [1] showed that the interference of the firmware while performing foreground tasks caused high performance fluctuations. One of the key aspects of a firmware driven SMR disk is the existence of a persistent cache at the center of the drive. Suppose we are benchmarking a fresh FS on a SMR drive, we might conceivably never hit the persistent cache limit (which is impossible as the drive actually ages), thus circumventing the large read-modify-write cycles that the firmware would have performed leading to low throughput.

The churn that the FS and the device endure while Geriatrix attempts to age according to an aging profile forces device aging as well, providing a more realistic aging effect.

Aging write-optimized FSs. Write-optimized FSs focus on expediting writes at the cost of possibly slower reads. Log-structured FSs [41] are a classic example of a write-optimized FS. Moreover, most modern storage devices operate as log-structured FSs internally, as mentioned in Section 3. In this architecture, every file rewrite causes file fragmentation because in-place updates are disallowed. Therefore, the true impact of aging a write-optimized FS is usually noticed when garbage collection starts interfering with foreground activity, a well studied problem also known as segment cleaning [41, 6, 28, 48]. These FSs are hard to age since they need to be forced into frequent garbage collection which is only possible at high space utilization and with significant free space fragmentation. Geriatrix fulfills these two requirements allowing for effective aging of write-optimized FSs.

Aging as a stress tester. An effective use of an aging exercise could be in the form of a stress tester. The high churn expected to be exercised by an aging tool can expose design flaws like overflows / underflows, data structure inefficiencies, concurrency and consistency issues among others. Geriatrix produces orders of magnitude more churn than the state of the art aging tools present today, rewriting data amounting to several times the specified FS image size. Thus, Geriatrix is as much a FS stress tester as it is an aging tool.

4 Geriatrix design and implementation

Geriatrix exercises a non-aged FS to match an *aging profile* which is provided as input, by performing a sequence of file create and delete operations. The profile contents are inspired by a combination of the usual parameters that affect a file's on-disk layout and which are easily obtainable from an aged instance of a FS using a single metadata tree walk. A Geriatrix aging profile comprises of:

- **FS fullness (bytes, %):** Partition size and fraction containing user data.
- **File size distribution (bytes, %; bytes, %; ...):** A histogram of file sizes.
- **Directory depth distribution (1, %, # subdirs; 2, % #subdirs; ...):** Path depth to individual files and percentage of files at that path depth along with the aggregate number of subdirectories at each depth.
- **Relative age distribution (n, %; m %; ...):** A histogram of relative file ages, $n < m < ...$, where younger files, in the first histogram bin make up the first % of all files in the aged FS image, and so on. More specifically, we extract create timestamps of files from an existing old FS snapshot following which we sort, enumerate and bin files into relative age buckets. These buckets approximate the age of each file relative to the other files, decoupling them from the absolute time of their creation (thus making their age unitless).

At a high level, Geriatrix aging proceeds in two distinct phases.

1. A ***rapid aging*** phase in which files are only created (one file creation per time instant or *tick*), to rapidly achieve the fullness target. Since this phase does not perform any deletions, there is no fragmentation induced in rapid aging. It merely achieves the required fullness while ensuring that the file size and directory depth distributions are met.
2. A ***stable aging*** phase in which each operation (one operation per tick) is either a file creation or a deletion based on a fair coin toss. This phase is designed to fit the relative age distribution while maintaining all other parameters. The roughly equal number of creations and deletions are necessary to maintain the fullness target, and in some sense mimic the steady-state operation of a FS operated at a certain fullness.

We now discuss how we ensure that the distributions of the file system being aged match the target distributions at the end of a Geriatrix run. We assume that all input distributions are mutually independent. Thus, we can easily achieve the size and directory depth distributions by drawing a size and directory depth value from their respective distributions and creating or deleting a file corresponding to those values. Note that rapid and stable aging phases continuously strive to maintain both, size and directory depth distributions.

However, achieving the target age distribution is harder. Note that the relative age of a file at the end of a Geriatrix run is the ratio of the number of ticks (or operations) that have passed since the file was created to the total number of ticks (say T) in that run. Thus, without knowing T, it is impossible to determine the final relative age bucket of any file. However, to direct the algorithm towards the target age distribution, it is necessary to know the final relative age bucket and thus, T. We overcome this by theoretically estimating a sufficiently large T within which it is possible to perform create / delete operations that achieve the target age distribution. Then, during the run, we use our estimate of T to compute the index of the final relative age bucket of any file; based on this, we perform clever deletions to ensure that the target age distribution is achieved. Appendix A provides this estimate of T and shows that when we stop the algorithm at T, it has indeed converged to the target age distribution.

Geriatrix has a repository of eight built-in file system aging profiles, most of which are from long-running file system and metadata publications [3, 13, 49, 33] to assist practitioners in conducting file system aging experiments. Table 3 provides a description of all the profiles along with the age of oldest file in every profile. We also indicate the wall-clock time it took to age these profiles in a ramdisk along with the total workload size generated during aging. Finally, we also show the empirical proof of our relative age convergence theorem via the perfect convergence (a root mean-squared error of $<$0.01%) achieved on the relative age distribution graphs for each aging profile (complete overlap in the graphs in Table 3).

The aging tool is a C++ program (built using the Boost library) designed to run on UNIX platforms. It has the ability to age both POSIX and non-POSIX FSs.

- **Reducing setup complexity:** Geriatrix is profile driven with eight built-in aging profiles. We also provide a repository of popular Linux FSs aged using the built-in profiles for standardized comparison.
- **Parallel aging:** Geriatrix has a configurable thread pool that exploits multi-threading in file systems to expedite aging substantially.
- **Reproducibility:** A user-defined seed governs all the randomness in Geriatrix, thus allowing every single-threaded Geriatrix execution to be exactly reproducible. For multi-threaded executions, the operating system scheduler may interleave threads differently resulting in different execution patterns across runs.
- **Rollback utility:** Aging experiments can take a prohibitively long time. Once a FS image has been aged, taking a snapshot of the image to be able to restore the same image for multiple tests is usually faster than re-aging. This requires a whole disk overwrite, which on today's multi-TB disks can take several hours, so we have developed a rollback utility to undo the effects of a short benchmark run on an aged image without having to replay the entire aged image again. Using the *blktrace* utility [7], we monitor the blocks that were modified during benchmark execution and effectively "undo" the perturbation caused by benchmarking by overwriting the dirtied blocks from the static snapshot of the aged image. *blktrace* adds overhead when running a benchmark, but is often negligible and can be mitigated further by writing the *blktrace* output to an in-memory FS or sending it across the network.
- **Multiple stopping conditions:** For many users, waiting for $< 0.01\%$ root mean square convergence of a Geriatrix run might be overkill. Thus, we have introduced multiple stopping conditions:
  1. the amount of time the ager is allowed to run
  2. the confidence [1] of the age distribution fit
  3. the max number of disk overwrites for aging

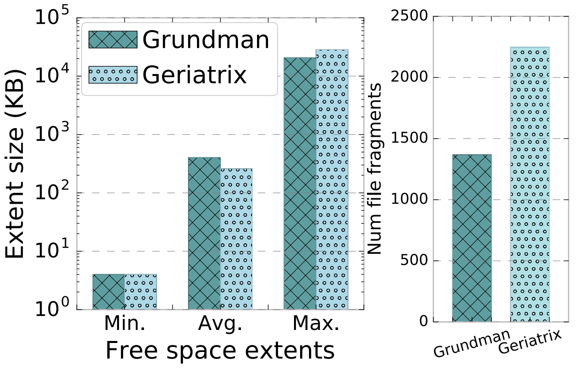
  Once any stopping condition is met, Geriatrix stops and displays the values of all three stopping conditions. The user can choose to accept the aging performed, or revise the condition(s) and resume aging.

5 Evaluation of Geriatrix as an aging tool

We evaluate the fidelity of Geriatrix's aging by comparing the file and free space fragmentation it induces on a fresh Ext4 partition to that of the source file system for the selected built-in profile. We do this comparison for two Geriatrix profiles: Grundman (extracted from a nine year old 90GB FS with approximately 90% fullness) and Dabre (extracted from a one year old 20GB FS with approximately 80% fullness). Despite being designed to only match externally visible measures of a FS, Geriatrix induces appropriate free space and file fragmentation by exercising the FS extensively.

We measure the distribution of free space extents using the *e2freefrag* utility. Figure 3 shows results for five file systems: the original Grundman FS image (aged naturally over 9 years), a fresh FS with no aging, a fresh FS aged by Geriatrix using the Grundman profile, a fresh FS

[1] Confidence of the convergence of distributions is calculated using the chi-squared goodness-of-fit statistic.



(a) Extents

(b) File Fragmentation

Figure 4: 4a compares the the minimum, average and maximum size of free space extents for a naturally aged Ext4 image and an Ext4 image aged using Geriatrix for the Grundman aging profile. 4b shows the number of fragments allocated as a result of a 2GB file being copied to both FS images.

aged by Impressions using the same file size distribution, and a fresh FS with the Grundman image files copied to it.

The primary takeaway is that the Geriatrix-aged FS matches the original Grundman FS closely, which can be seen by comparing the colorful bars, while the other three (gray bars) do not. As expected, the freshly formatted Ext4 has very large free space extents, mostly between 1-2GB. Grundman and the Geriatrix-aged FS have much smaller extents and more spread out free space extent distributions ranging from 4KB to 32MB. The "Copied" and Impressions-aged FSs are similar to the freshly-formatted file system, with low free space fragmentation. The comparison using the Dabre profile (graph omitted due to space constraints) shows very similar results.

Figure 4a compares the minimum, average and maximum free space extent sizes for Grundman and the Geriatrix-aged FSs. Both have the same smallest free space extent of 4KB. The average free space extent size of naturally-aged Grundman is only 144KB larger than its Geriatrix counterpart, while the largest free space extent is only 7.2MB smaller. These numbers are very close considering the total partition size of 90GB.

Figure 4b measures the fragmentation of a new 3GB file copied to each of the naturally-aged Grundman image and the Geriatrix-aged FS, using the *filefrag* utility. The image aged by Geriatrix splits the file into 2250 fragments while the naturally-aged FS only splits it into 1368 fragments. Despite Geriatrix over-splitting the file, its aging is two orders of magnitude higher than the number of fragments created writing 3GB to a freshly formatted Ext4.

Since Geriatrix refrains from taking shortcuts, and performs millions of operations before declaring a FS *aged*, it closely approximates the FS state caused by natural aging. The Geriatrix aging experiment reported in Figures 3 and 4 took approximately 420 minutes. Recreating the fragmentation naturally occurring in nine years with only 420 minutes of aging is an acceleration of >11000×.

6 How Geriatrix changes conclusions

To highlight the impact of aging, we recreated experiments from Btrfs [39], F2fs [24] and NOVA [52] publications on unaged and aged FS instances. We also produced aged instances of Ext4 (used for comparison across all three papers) and Xfs (used for comparison in the Btrfs and NOVA papers).

Experimental setup. We performed all our experiments on an Emulab PRObE cluster [15]. The hardware used and the setups for experiments is described in Table 2. For fairer comparison, we matched the memory and the number of cores in our benchmark when recreating expts. from the Btrfs [39] and F2fs [24] publications.

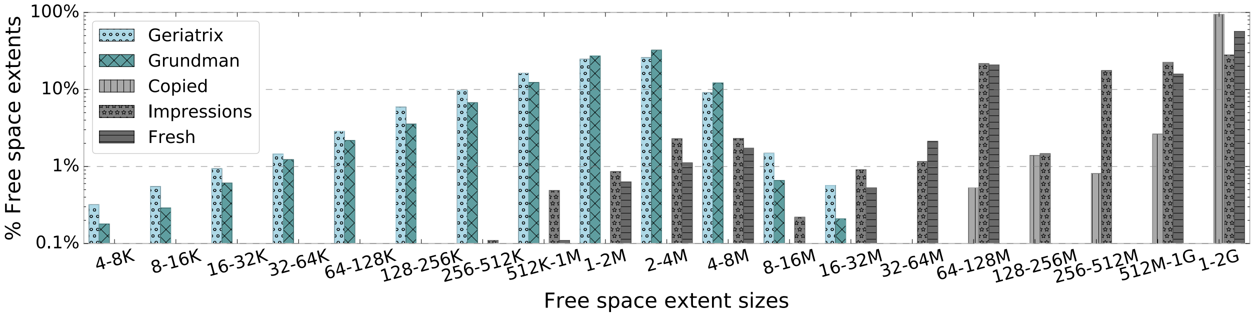


Figure 3: Free space fragmentation comparison of an actual old Ext4 FS image (Grundman) with Geriatrix driven by the Grundman profile, Impressions driven by the Grundman profile's file size distribution, a partition with the contents of the original Grundman image copied over and a freshly formatted Ext4 partition. Other than the freshly formatted image, all other FS images are approximately 90% full. Geriatrix induces free space fragmentation very similar to the naturally aged Grundman image with no large free space extents, hence causing appropriate free space fragmentation.

Paper	Disk	RAM	CPU (cores)	Linux (Kernel Version)
Btrfs [39]	500GB HDD (WDC WD5000YS-01MPB0)	2GB	Intel Xeon E7 (8)	Ubuntu 14.04 LTS (3.13.0-33)
	64GB SSD (Crucial M4-CT064M4SSD2)	2GB	AMD Opteron (8)	Ubuntu 14.04 LTS (3.13.0-33, 4.4.0-31)
F2fs [24]	64GB SSD (Crucial M4-CT064M4SSD2)	4GB	Intel Core i7 (4)	Ubuntu 14.04 LTS (4.4.0-31)
	120GB SSD (ADATA SSD S510)	4GB	Intel Core i7 (4)	Ubuntu 14.04 LTS (4.4.0-31)
NOVA [52]	64GB NVM (Emulated in DRAM)	8GB	AMD Opteron (8)	Ubuntu 14.04 LTS (4.13)

Table 2: Experimental Configuration.

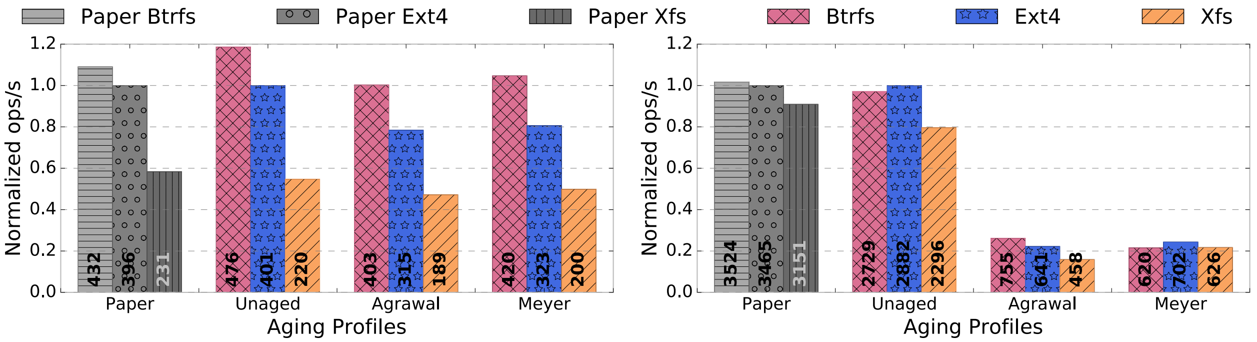
We used four of Geriatrix's built-in profiles in our experiments: Agrawal [3], Meyer [33], Dabre and Pramod. We performed aging in memory and captured the resulting aged images. We consciously decided to not age the FSs on the device as we wanted to prevent device aging from affecting the FS aging. Prior to each benchmark run we copied the corresponding aged image onto a disk (using *dd* to the raw device) and mounted the FS on the aged image. All FSs were mounted using default mount options.

The Filebench benchmark [30] was used for all performance measurements with different profiles according to the appropriate reference publication. The primary performance metric reported is *overall operations per second* as reported by Filebench. Each benchmark run lasted about 10 minutes and we performed three runs of each benchmark to capture variance. We report only the mean, since the maximum standard deviation observed was below 2. Since our hardware is not identical to what was used in the papers and since we are testing with code potentially newer than the one used for publication, exact reproduction of paper results even for unaged instances of FSs is unlikely. With SSDs, the performance variability across devices is especially high. For ease of comparison, we include raw data on the bar graphs, but normalize bar heights. The published results (leftmost gray bars) are normalized to the published Ext4 results, and the aged FS performance numbers are normalized to unaged Ext4 performance on the same hardware. We chose Ext4 because it is the default FS rolled out with most Linux distributions today. All HDD experiments were conducted using 100GB aged images with a 80% capacity utilization target being replayed on a 100GB partition of a 500GB HDD.

HDD experiments. Figure 2a in Section 1 compares the performance of Btrfs, Ext4 and Xfs on an HDD for the Filebench varmail workload, representing a mail server workload of tiny (≈16KB) file operations. After aging using the Meyer profile, we see 9-30% slowdowns, with Btrfs being most affected by aging, followed by Ext4 and then Xfs.

Figure 2b compares the same FSs using the webserver workload. The webserver workload is highly multi-threaded and again, operates on tiny files, although it avoids issuing expensive fsync operations and mimics webservers that have to perform more whole-file reads and log appends. Since FSs are usually more sensitive to small file operations, it is perhaps understandably harder to reproduce published results; indeed, unaged Btrfs performs 22% faster on our hardware than reported in the paper, while unaged Xfs also performs 10% faster than the paper. We also see a minor inversion of ranking, with unaged Btrfs outperforming unaged Xfs. The performance penalties after aging are between 12-17%. We show only the Meyer aging profile in webserver and varmail because Btrfs could not sustain aging for the Pramod profile, and although Btrfs successfully completed aging for Dabre and Agrawal profiles, it did not complete execution of the benchmark despite having the required space to do so. This exemplifies the role of Geriatrix as a stress testing tool.

Figure 5a compares the FSs on the fileserver workload,



(a) Btrfs HDD Fileserver

(b) Btrfs SSD Fileserver

Figure 5: Both graphs reproduce Filebench fileserver experiments from the Btrfs ACM TOS publication [39] on aged FS instances on a HDD (Figure 5a) and a SSD (Figure 5b). Aged Btrfs and Ext4 performed at most 22% slower on the HDD but supported the prior paper's published rank ordering whereas aged Btrfs and Ext4 on a SSD degraded benchmark performance by as much as 80%, and changed the rank ordering of compared FSs. SSD experiments in 5b were performed using the Linux kernel version 3.13.0-33

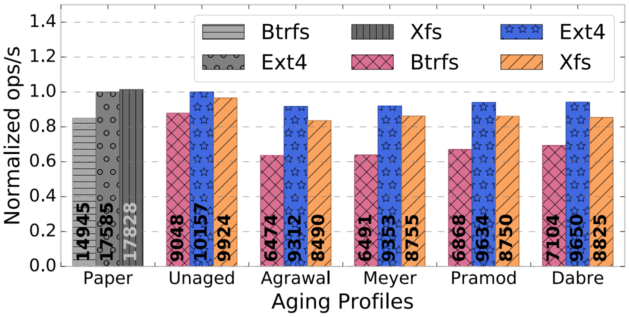


Figure 6: Filbench webserver recreations from the Btrfs paper [33] on SSD. Btrfs is most affected by aging with its performance dropping by 30%. Ext4 and Xfs performance drops by a maximum of 10% and 15% respectively.

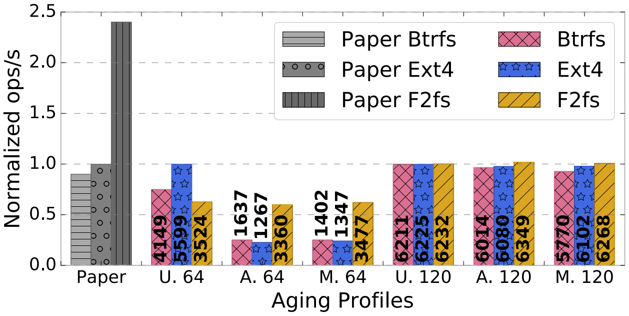


Figure 7: Recreation of the Filebench fileserver benchmark from the F2fs paper [24] on two SSDs - 64GB (labeled 64) and 120GB (labeled 120). Performance on 64 drops significantly after aging, especially for Btrfs and Ext4 resulting in a graph that looks similar to published results. Refer Section 6 for detailed explanation. 120 seems unaffected by aging, thus highlighting highly varied performance across different SSDs

which consists of relatively larger file writes and reads (≈ 128KB) compared to thousands of tiny file operations in webserver and varmail. We observe that unaged Btrfs is 10% faster in our setup, unaged Ext4 is marginally better while unaged Xfs is about 5% slower, keeping performance similar to published results with slightly increased performance gaps between the FSs. After aging using the Agrawal profile, we observe a 10-22% performance drop with Ext4 being the most affected FS.

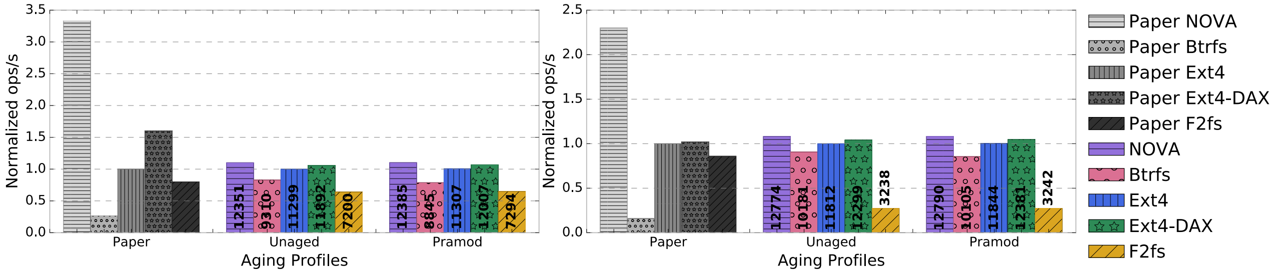
SSD experiments. The SSD experiments were conducted on a 64GB SSD with a 59GB aged FS image with a 70% fullness target. The reason for choosing 70% was to allow the benchmarking workload to fit after aging. SSDs are available in a variety of product price-point classes and have highly variable performance making reproduction of SSD results on different hardware unlikely. This is evident from figures 2c and 5b which show a completely different rank ordering of unaged Btrfs, Ext4 and Xfs compared to published results on the fileserver workload. While Btrfs had the best performance in the paper, it ranged from being the best in aged reproduction using the Dabre (Figure 2c) and Agrawal profiles to being the worst in the aged reproduction using the Meyer profile. Apart from the rank ordering, we observed massive post-aging slowdowns on SSDs from 65-80%. As explained in Sections 1 and 3, we attribute this performance drop to SSD device aging along with FS aging, with both effects being exaggerated by the free space fragmentation caused by Geriatrix.

The webserver results shown in Figure 6 also show significant slowdowns, but are not so dramatic. Btrfs appears to be the most affected by aging, showing up to a 30% performance drop, while Xfs and Ext4 degrade by ≤15% and ≤10%, respectively.

It was typical of SSDs from a few years ago to not be able to sustain more than 2 minutes of continuous writing before performing inline cleaning [37]. Our benchmarking technique involves writing an aged image on almost the entire surface of the SSD, performing a 10 minute benchmark run followed by unmounting the FS and repeating the process with 100% device utilization. An entire surface rewrite should be equivalent to a giant trim obviating the need to perform any internal garbage collection in the FTL, but this is dependent on firmware implementation which varies widely across devices.

Figure 7 is the recreation of F2fs [24] results on SSDs comparing Btrfs, Ext4 and F2fs using the Filebench fileserver profile. To capture variability of performance across devices, we chose SSDs of different makes and sizes - a 64GB Crucial SSD with 59GB aged FS images (bars labeled 64) and a 120GB ADATA SSD with 100GB aged FS images (bars labeled 120). Ext4 is the winning FS when comparing unaged FS instances on 64GB SSD, and Xfs is marginally better on the 120GB SSD, while published results report F2fs performance was 2.4× that of Ext4. Aging on 64GB SSD shows interesting behavior as the performance of all three FSs drops (61-67% for Btrfs, 76-78% for Ext4 and 2-5% for F2fs) and the outcome looks similar to results that the earlier paper reported. The authors most likely aged the SSD firmware by performing repeated benchmark runs resulting in behavior similar to what is seen when FSs are aged. In contrast, the 100GB FSs on the 120GB SSD age much more gracefully with only Btrfs showing as much as 7% performance penalty after aging. This again suggests that SSDs themselves age in different (and non-trivial) ways along with the FSs running on them.

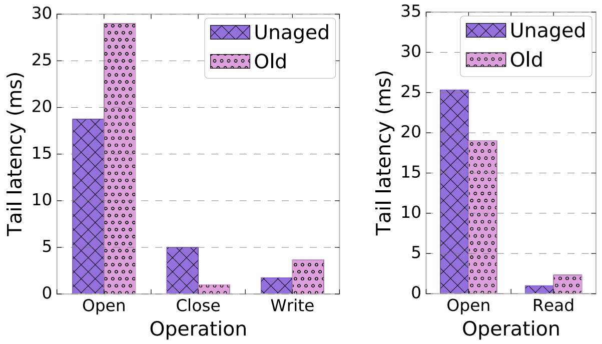
Emulated NVM experiments. We also perform aging experiments on NOVA [52] – a log-structured FS intended for non-volatile memory (NVM). We used a modified Linux kernel version 4.13 to age NOVA since it required special kernel libraries for enabling persistent memory emulation. A 64GB partition similar to the SSD experiments was aged with 70% utilization. The NOVA paper performed experiments with 64GB persis-



(a) NOVA Fileserver

(b) NOVA Varmail

Figure 8: Both graphs reproduce experiments from the NOVA FAST 2016 publication [52] on aged FS instances using emulated non-volatile memory. All FSs age gracefully with Btrfs seeing the largest performance hit of 5% on the fileserver workload (Figure 8a) and 6% on the varmail workload (Figure 8b). Graphs from the paper are for reference and in this case cannot be compared to our reproductions because of different hardware and configuration.



(a) NOVA Fileserver Latency

(b) NOVA Varmail Latency

Figure 9: 9a shows the latency of the slowest operations for the fileserver workload. Aged NOVA slows down by upto 60% on file opens and upto 2× on file writes. In 9b we see slowdowns of upto 2.3× on reads of the slowest file in the varmail workload. The aged opens are contrastingly faster for the varmail workload compared to the fileserver workload. Slowest file open during the varmail workload run is 25% faster after aging.

tent memory devoted to NOVA and 32GB DRAM for the rest of the system. The experiment dataset size was made slightly larger than DRAM (more than 32GB) forcing NVM device interaction during an experiment run. However, that meant that the authors used more than half the 64GB NVM device for their benchmark run. Generally, a <50% utilized FS would be considered too underutilized for running a realistic post-aging benchmark. Hence, we kept utilization at 70%, reduced the DRAM size to 8GB and exercised a workload of more than 8GB to still ensure that the benchmarking workload was larger than the system memory. Furthermore, the authors performed their experiments on special Intel NVM hardware. Thus, we discourage the direct comparison of performance in Figures 8a and 8b with the NOVA paper results.

Figure 8a compares NOVA's performance on the fileserver workload before and after aging with Btrfs, Ext4, Ext4-DAX and F2fs. Ext4-DAX is Ext4 mounted with the *-o dax* option to enable direct-access to the emulated NVM device bypassing the buffer cache, thus avoiding duplicate caching of data. With the largest difference of 5% observed in Btrfs, we see virtually no difference in all FSs before and after aging. The same is true in the case of the varmail workload shown in Figure 8b where Btrfs is the most affected FS with a slowdown of approximately 6%.

Although aging does not affect throughput, its effect is seen in the tail latencies. Figure 9a shows the highest latency encountered by Filebench categorized by operation. The slowest file open in the fileserver workload was 60% slower after aging. Writing the slowest file also took twice as long compared to a freshly formatted NOVA image. Surprisingly, closing the slowest file was 5× faster after aging. In the varmail workload, the slowest aged read was 2.3× slower than the slowest unaged read. In contrast, the opening of the slowest aged file was approximately 25% faster. Both these observations can be attributed to the log-structured design of NOVA, as log-structured FSs are not read-optimized. We speculate that reorganization of files after cleaning might have led to the open call being executed faster.

As expected, with no mechanical parts and DRAM-like latency, NVM or in-memory FSs do not show significant reduction in throughput after aging. An aging exercise for these FSs is mainly about exposing inefficiencies in FS implementations that usually get hidden behind massive device latencies.

7 Conclusion

The Geriatrix aging tool creates representative fragmentation of both files and free space. Unlike with HDDs, file system performance on SSDs is greatly impacted by free space fragmentation that has been largely absent from prior aging approaches. Being released as open-source, Geriatrix will enable file system benchmarking to include aging relevant to modern storage.

8 Acknowledgements

We thank Keith Smith, Rob Johnson and the anonymous reviewers for their feedback and guidance. We also thank the members and companies of the PDL Consortium (Alibaba, Broadcom, Dell EMC, Facebook, Google, Hewlett-Packard Enterprise, Hitachi, Intel, IBM, Micron, Microsoft Research, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Toshiba, Two Sigma, Veritas and Western Digital) for their interest, insights, feedback, and support.

A Proof of convergence of the age distr.

Assume we need to age a FS that has K files at the end of the rapid aging phase, and continues to have approximately K files throughout aging (i.e. one Geriatrix run). Let T be the total number of operations (each corresponding to a tick) performed in one Geriatrix run. Let the target relative age distribution be specified by binning the T ticks into B buckets, with the oldest bucket indexed as $b = B$ and the youngest bucket as $b = 1$ (i.e., the files created at the end of the run will fall in $b = 1$). Let s_b be the relative size of the age bucket b, i.e., files created in the first Ts_B operations will fall in relative age bucket B, and those in the next Ts_{B-1} operations in bucket $B-1$ and so on). Let g_b be the relative number of files in bucket b according to the target distribution i.e., there must be approximately Kg_b files in bucket b at the end of a run. Then we can predict the number of operations required to achieve convergence as follows:

Theorem A.1. *After*

$$T = max \begin{cases} \frac{2Kg_b}{s_b}, \forall b < B \\ \\ \frac{K}{s_B} \end{cases} .$$

the Geriatrix run would have converged to the target age distribution i.e., the number of files in age bucket b would equal Kg_b (approximately).

Proof. First, let us examine how the rapid aging phase, which consists of the first K file creations, affects the age distribution of the system. Since the oldest bucket corresponds to the first Ts_B operations, and $Ts_B \geq K$, all the files created in the rapid aging phase end up falling in bucket B.

Next, we will examine how the stable aging phase can achieve the target age distribution. Let O_b be the total number of stable aging operations that correspond to the ticks corresponding to bucket b. For the oldest bucket, $O_B = Ts_B - K$ and in the other buckets $O_b = Ts_b$. Further, let C_b and D_b respectively be the number of stable aging creation and deletion operations performed corresponding to bucket b. Assume that $C_b \approx \frac{O_b}{2}$ and $D_b \approx \frac{O_b}{2}$ since we perform creations and deletions in the stable aging phase using a fair coin toss.

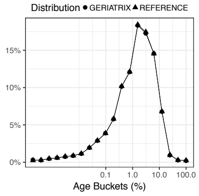
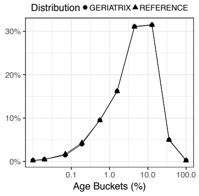
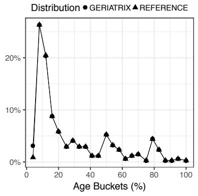
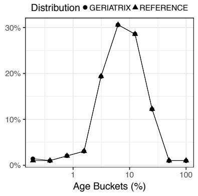
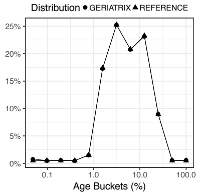
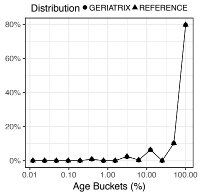
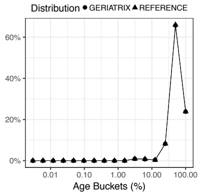
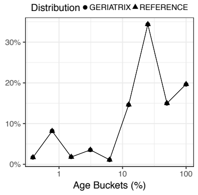
Now note that the C_b creations corresponding to bucket b will create files which will all fall in bucket b. On the other hand, the D_b deletions corresponding to bucket b may be performed either on files in bucket b or on any pre-existing file in an older bucket b' such that $b' > b$. We will show that we can distribute these deletions in such a manner that the target age distribution of Kg_b files in bucket b can be achieved for every b.

First, observe that in the youngest bucket $b = 1$ we create exactly $C_1 = Ts_1/2$ files. Now, since $T \geq 2Kg_1/s_1$, $C_1 \geq Kg_1$. Thus, there would be an excess of $C_1 - Kg_1 = Ts_1/2 - Kg_1$ file creations in this bucket which need to be deleted. Since we have $D_1 = Ts_1/2$ deletion operations available for this bucket, we can use $Ts_1/2 - Kg_1$ of them to remove these excess files and achieve the target for this bucket, and use the excess Kg_1 delete operations for the older buckets.

We will now use induction to show that we can similarly achieve the target number of files for the older buckets (except the oldest which we will handle separately). Specifically, assume that for operating on the files in bucket b where $b \neq B$, we have $K\sum_{b'=1}^{b-1} g_{b'}$ excess delete operations available from the operations corresponding to the younger buckets (this is equal to zero for $b = 1$). Next, recall that have $C_b = Ts_b/2$ file creations in this bucket. But since $T \geq 2Kg_b/s_b$, $C_b \geq Kg_b$. Thus, there would be an excess of $Ts_b/2 - Kg_b$ files in this bucket that need to be deleted. However, we have $Ts_b/2$ deletion operations available from the ticks corresponding to this bucket and a further $K\sum_{b'=1}^{b-1} g_{b'}$ excess deletion operations available from younger buckets. Hence, we can delete these excess $Ts_b/2 - Kg_b$ files to achieve the target of Kg_b files in this bucket. Then, we will have the remaining $Kg_b + K\sum_{b'=1}^{b-1} g_{b'} = K\sum_{b'=1}^{b} g_{b'}$ deletion operations as excess which can be used for the older buckets, thus satisfying the induction hypothesis.

Now, for the oldest bucket we will have $K\sum_{b'=1}^{B-1} g_{b'} = K(1-g_B)$ excess deletion operations available (since $\sum_{b'=1}^{B} g'_b = 1$) from the younger buckets, besides $Ts_B/2$ deletions corresponding to this bucket. At the same time, we have created K files in this bucket in the rapid aging phase and $Ts_B/2$ files in the stable aging face i.e., $K + Ts_B/2$ files in total. Hence, we will have an excess of $K + Ts_B/2 - Kg_B = K(1-g_B) + Ts_B/2$ files, which happens to be equal to the total number of deletions available. Hence, we can achieve the target file number in this bucket *while using up all the available operations*. Thus, we use exactly T operations to achieve the target age distribution.

□

Profile	*Description*	*Age (yrs)*	*Duration (min)*	*Overwrites (50 GB)*	*Age Distribution*
Douceur*	Referenced from a study of FS contents by Douceur et al. [13] in 1998. It captures an aggregate analysis of over 10000 commercial PCs running Microsoft Windows.	4	NA	22422 (1 GB)	
Agrawal	Referenced from a metadata study of Windows desktop FSs from Microsoft in 2004 [3]. Most computers ran NTFS (80%) along with FAT32 (15%) and FAT (5%).	14	466	253	
Meyer	Referenced from a deduplication study conducted on 857 Windows desktop computers at Microsoft [33]. Snapshots of the FSs were taken in 2009.	2	78	159	
Wang-OS	Referenced from an HPC FS environment study [49] performed on NetApp's WAFL [16] installations at CMU's Parallel Data Lab (an educational cluster setup for systems' research) in 2011.	22	231	34	
Wang-LANL	Referenced from the same study as Wang-OS [49] from Panasas FS [51] installations at Los Alamos National Lab (LANL).	11	146	28	
Dabre	Captured in 2017 from the root partition of a colleague's laptop running Ext4.	1	91	4042	
Pramod	Captured in 2017 from the root partition of a colleague's laptop running Ext4.	3.75	27	17	
Grundman	Captured in 2018 from the home partition of a colleague's laptop running Ext4.	8.75	142	2388	

*Douceur 1 GB image required a 22.4 TB workload to converge, thus taking too long to converge for 50 GB.

Table 3: List of built-in aging profiles in Geriatrix with their descriptions, the age of the oldest file in each profile, the duration to age a 50GB Xfs partition in memory with Geriatrix for every profile, and the number of disk overwrites (workload) performed.

References

[1] AGHAYEV, A., SHAFAEI, M., AND DESNOYERS, P. Skylighta window on shingled disk operation. *ACM Transactions on Storage (TOS)* (2015).

[2] AGRAWAL, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Generating realistic impressions for file-system benchmarking. *ACM Transactions on Computer Systems (TOCS)* (2009).

[3] AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)* (2007).

[4] ARANYA, A., WRIGHT, C. P., AND ZADOK, E. Tracefs: A file system to trace them all. In *USENIX File and Storage Technologies (FAST)* (2004).

[5] BATSAKIS, A., BURNS, R., KANEVSKY, A., LENTINI, J., AND TALPEY, T. Ca-nfs: A congestion-aware network file system. *ACM Transactions on Storage (TOS)* (2009).

[6] BLACKWELL, T., HARRIS, J., AND SELTZER, M. I. Heuristic cleaning algorithms in log-structured file systems. In *USENIX Annual Technical Conference (ATC)* (1995).

[7] BRUNELLE, A. D. Block i/o layer tracing: blktrace. *HP, Gelato-Cupertino, CA, USA* (2006).

[8] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. A study of integrated prefetching and caching strategies. *ACM SIGMETRICS Performance Evaluation Review* (1995).

[9] CIPAR, J., CORNER, M. D., AND BERGER, E. D. Tfs: A transparent file system for contributory storage. In *USENIX File and Storage Technologies (FAST)* (2007).

[10] COMER, D. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)* (1979).

[11] CONWAY, A., BAKSHI, A., JIAO, Y., JANNEN, W., ZHAN, Y., YUAN, J., BENDER, M. A., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., ET AL. File systems fated for senescence? nonsense, says science! In *USENIX File and Storage Technologies (FAST)* (2017).

[12] CONWAY, A., BAKSHI, A., JIAO, Y., ZHAN, Y., BENDER, M. A., JANNEN, W., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., YUAN, J., ET AL. How to fragment your file system. *USENIX ;login:* (2017).

[13] DOUCEUR, J. R., AND BOLOSKY, W. J. A large-scale study of file-system contents. *ACM SIGMETRICS Performance Evaluation Review* (1999).

[14] FEIERTAG, R. J., AND ORGANICK, E. I. The multics input/output system. In *ACM Symposium on Operating Systems Principles (SOSP)* (1971).

[15] GIBSON, G., GRIDER, G., JACOBSON, A., AND LLOYD, W. Probe: A thousand-node experimental cluster for computer systems research. *USENIX ;login:* (2013).

[16] HITZ, D., LAU, J., AND MALCOLM, M. A. File system design for an nfs file server appliance. In *USENIX Winter Conference* (1994).

[17] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)* (1988).

[18] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., ET AL. Betrfs: A right-optimized write-optimized file system. In *USENIX File and Storage Technologies (FAST)* (2015).

[19] JOSEPHSON, W. K., BONGO, L. A., LI, K., AND FLYNN, D. Dfs: A file system for virtualized flash storage. *ACM Transactions on Storage (TOS)* (2010).

[20] JOUKOV, N., WONG, T., AND ZADOK, E. Accurate and efficient replaying of file system traces. In *USENIX File and Storage Technologies (FAST)* (2005).

[21] KONISHI, R., AMAGAI, Y., SATO, K., HIFUMI, H., KIHARA, S., AND MORIAI, S. The linux implementation of a log-structured file system. *ACM Operating Systems Review (SIGOPS)* (2006).

[22] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A cross media file system. In *ACM Symposium on Operating Systems Principles (SOSP)* (2017).

[23] LE, D., HUANG, H., AND WANG, H. Understanding performance implications of nested file systems in a virtualized environment. In *USENIX File and Storage Technologies (FAST)* (2012).

[24] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2fs: A new file system for flash storage. In *USENIX File and Storage Technologies (FAST)* (2015).

[25] LEE, S., LIU, M., JUN, S., XU, S., KIM, J., ET AL. Application-managed flash. In *USENIX File and Storage Technologies (FAST)* (2016).

[26] LU, Y., SHU, J., AND WANG, W. Reconfs: A reconstructable file system on flash storage. In *USENIX File and Storage Technologies (FAST)* (2014).

[27] MASON, C. Compilebench. https://oss.oracle.com/ mason/compilebench.

[28] MATTHEWS, J. N., ROSELLI, D., COSTELLO, A. M., WANG, R. Y., AND ANDERSON, T. E. Improving the performance of log-structured file systems with adaptive methods. In *ACM Symposium on Operating Systems Principles (SOSP)* (1997).

[29] MAZUREK, M. L., THERESKA, E., GUNAWARDENA, D., HARPER, R. H., AND SCOTT, J. Zzfs: a hybrid device and cloud file system for spontaneous users. In *USENIX File and Storage Technologies (FAST)* (2012).

[30] MCDOUGALL, R., AND MAURO, J. Filebench. http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf, 2005.

[31] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)* (1984).

[32] MESNIER, M. P., WACHS, M., SIMBASIVAN, R. R., LOPEZ, J., HENDRICKS, J., GANGER, G. R., AND O'HALLARON, D. R. //trace: Parallel trace replay with approximate causal events. In *USENIX File and Storage Technologies (FAST)* (2007).

[33] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. *ACM Transactions on Storage (TOS)* (2012).

[34] MIN, C., KIM, K., CHO, H., LEE, S.-W., AND EOM, Y. I. Sfs: random write considered harmful in solid state drives. In *USENIX File and Storage Technologies (FAST)* (2012).

[35] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *ACM Symposium on Operating Systems Principles (SOSP)* (1995).

[36] PETROVIC, S. The effects of age on file system performance. Master's thesis, Masaryk University, 5 2017.

[37] POLTE, M., SIMSA, J., AND GIBSON, G. Enabling enterprise solid state disks performance. *Workshop on Integrating Solid-state Memory into the Storage Hierarchy* (2009).

[38] RITCHIE, O., AND THOMPSON, K. The unix time-sharing system. *The Bell System Technical Journal* (1978).

[39] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)* (2013).

[40] ROSELLI, D. S., LORCH, J. R., ANDERSON, T. E., ET AL. A comparison of file system workloads. In *USENIX Annual Technical Conference (ATC)* (2000).

[41] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* (1992).

[42] SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison. In *USENIX Technical Conference Proceedings (TCON)* (1995).

[43] SMITH, K., AND SELTZER, M. I. File layout and file system performance. *Tech. Rep. TR-35-94, Harvard University* (1994).

[44] SMITH, K. A., AND SELTZER, M. I. File system aging increasing the relevance of file system benchmarks. In *ACM SIGMETRICS Performance Evaluation Review* (1997).

[45] THOMPSON, D. A., AND BEST, J. S. The future of magnetic data storage techology. *IBM Journal of Research and Development* (2000).

[46] UNGUREANU, C., ATKIN, B., ARANYA, A., GOKHALE, S., RAGO, S., CALKOWSKI, G., DUBNICKI, C., AND BOHRA, A. Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system. In *USENIX File and Storage Technologies (FAST)* (2010).

[47] VRABLE, M., SAVAGE, S., AND VOELKER, G. M. Bluesky: a cloud-backed file system for the enterprise. In *USENIX File and Storage Technologies (FAST)* (2012).

[48] WANG, J., AND HU, Y. Wolf-a novel reordering write buffer to boost the performance of log-structured file systems. In *USENIX File and Storage Technologies (FAST)* (2002).

[49] WANG, Y. A statistical study for file system meta data on high performance computing sites. Master's thesis, Southeast University, 2012.

[50] WEISS, Z., HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Root: Replaying multithreaded traces with resource-oriented ordering. In *ACM Symposium on Operating Systems Principles (SOSP)* (2013).

[51] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G. A., MUELLER, B., SMALL, J., ZELENKA, J., AND ZHOU, B. Scalable performance of the panasas parallel file system. In *USENIX File and Storage Technologies (FAST)* (2008).

[52] XU, J., AND SWANSON, S. Nova: a log-structured file system for hybrid volatile/non-volatile main memories. In *USENIX File and Storage Technologies (FAST)* (2016).

[53] YUAN, J., ZHAN, Y., JANNEN, W., PANDEY, P., AKSHINTALA, A., CHANDNANI, K., DEO, P., KASHEFF, Z., WALSH, L., BENDER, M., ET AL. Optimizing every operation in a write-optimized file system. In *USENIX File and Storage Technologies (FAST)* (2016).

[54] ZHANG, S., CATANESE, H., AND WANG, A. A.-I. The composite-file file system: decoupling the one-to-one mapping of files and metadata for better performance. In *USENIX File and Storage Technologies (FAST)* (2016).

[55] ZHANG, Z., AND GHOSE, K. yfs: A journaling file system design for handling large data sets with reduced seeking. In *USENIX File and Storage Technologies (FAST)* (2003), vol. 3, p. 2nd.

[56] ZHU, B., LI, K., AND PATTERSON, R. H. Avoiding the disk bottleneck in the data domain deduplication file system. In *USENIX File and Storage Technologies (FAST)* (2008).

[57] ZHU, N., CHEN, J., CHIUEH, T.-C., AND ELLARD, D. Tbbt: scalable and accurate trace replay for file server evaluation. In *ACM SIGMETRICS Performance Evaluation Review* (2005).

Can't We All Get Along?
Redesigning Protection Storage for Modern Workloads

Yamini Allu Fred Douglis* Mahesh Kamat Ramya Prabhakar Philip Shilane Rahul Ugale

Dell EMC

Abstract

Deduplication systems for traditional backups have optimized for large sequential writes and reads. Over time, new applications have resulted in nonsequential accesses, patterns reminiscent of primary storage systems. The Data Domain File System (DDFS) needs to evolve to support these modern workloads by providing high performance for nonsequential accesses without degrading performance for traditional backup workloads.

Based on our experience with thousands of deployed systems, we have updated our storage software to distinguish user workloads and apply optimizations including leveraging solid-state disk (SSD) caches. Since SSDs are still significantly more expensive than magnetic disks, we make our system cost-effective by caching metadata and file data rather than moving everything to SSD. We dynamically detect access patterns to decide when to cache, prefetch, and perform numerous other optimizations. We find that on a workload with nonsequential accesses, with SSDs for caching metadata alone, we measured a 5.7× improvement on input/output operations per second (IOPS) when compared to a baseline without SSDs. Combining metadata and data caching in SSDs, we measured a further 1.7× IOPS increase. Adding software optimizations throughout our system added an additional 2.7× IOPS improvement for nonsequential workloads. Overall, we find that both hardware and software changes are necessary to support the new mix of sequential and nonsequential workloads at acceptable cost. Our updated system is sold to customers worldwide.

1 Introduction

With traditional backups, an application periodically copies the entire contents of a file system into the backup environment, with changes since the last copy added at shorter intervals. These are called *full* and *incremental* backups, respectively [8]. Deduplicating file systems leverage the redundancy across full backups by storing a single copy of data, with the granularity of duplicate detection varying from whole files [2] to individual file blocks [31] or variable-sized "chunks" that are determined on the fly via content characteristics [28, 36]. Leveraging a log-structured file system [32] to store nonduplicate data results in append-only write operations. Nonsequential reads are needed for index lookups and when deduplication results in the physical fragmentation of unique data [20]. However, index lookups can be limited to trade deduplication efficiency to improve performance both for writes and reads by writing some duplicates to improve data locality [13].

There have been recent reports about the impact of evolving workloads on system performance. An article [3] provided an overview of the impact of increasing numbers of small files and higher deduplication ratios (and other changing properties) on the Data Domain File System (DDFS) as a whole, but with relatively few details or quantitative analysis. As an example, garbage collection (GC) was slowed by these changing workloads and a new algorithm was needed [11].

Indeed, GC is not the only aspect of the system that must be rethought to handle modern workloads. While Data Domain was one of the original "purpose-built backup appliances," modern data protection workloads impose very different requirements. These workloads include frequent updates to arbitrary locations in backup files, direct access to individual files rather than the aggregates created by traditional backup applications, and direct read/write access to files in the appliance by applications on other hosts. This last class of usage is particularly demanding, as it generally involves large amounts of **nonsequential I/O (NSIO)**.

Thus, we are at an inflection point where we need to rethink and redesign backup systems to enable *optimized*

[*] Current affiliation: Perspecta Labs

performance for non-traditional data protection workloads with nonsequential accesses for our customers. This goes beyond previous work on improving deduplicating storage by reducing fragmentation of sequentially read data [20], as it strives to provide improved NSIO performance without degrading the performance of traditional, sequential, workloads. These two types of application must coexist regardless of the distribution of workloads between the two categories.

In this paper, we describe the evolution of DDFS to support both traditional and nontraditional workloads based on our experience with deployed systems. Traditional workloads mostly have large sequential writes and reads with low metadata operations. Nontraditional workloads have many small files, more metadata operations, and frequent nonsequential accesses. Our new DDFS design supports higher IOPS for both metadata operations and nonsequential reads and writes and is already used by our customers[1]

We expanded our storage system, which had used only hard disk drives (HDDs), to also use solid-state disks (SSDs). These cache index data, directory structures, deduplicating file recipes, and ultimately file data. Because the capacity of the SSDs is a small fraction of the overall system (e.g., 1%), we must make a number of tradeoffs. For instance, while the index that maps fingerprints to disk location stores information in the SSDs for all chunks, we use a shorter form of the fingerprint that can have occasional hash collisions. We rely on metadata accessed when reading a chunk to provide the full fingerprint for confirmation, and when a mismatch is detected, the full on-disk index is consulted.

In addition, we have made several changes to our software stack. These include dynamic assessment of prefetching and caching behavior based on access patterns; data alignment, using application-specific chunk size; scheduler changes for quality of service; increasing the parallelism of I/O requests to a single file; minimizing writes by queuing metadata updates in memory; and support for smaller access sizes.

Through lab experiments, we demonstrate the impact of these changes on the performance of certain applications. With a NSIO workload, with SSDs for caching metadata, we measured a 5.7× IOPS improvement relative to a system without SSDs. Adding data caching in SSDs, we measured a further 1.7× IOPS increase. Combining SSD caching with software optimizations throughout our system added an additional 2.7× IOPS increase for NSIO workloads. We measured similar factors of reductions in terms of average latency of accesses for both reads and writes. Importantly, performance for traditional workloads running concurrently remained high and even increased (when run separately) due to software improvements. We provide detailed experimental results in §6.

In summary, the main contributions of this paper are:

1. We extend support to modern backup applications, with NSIO access patterns. We ensure our new design for the DDFS software stack benefits both traditional and nontraditional backup and restore tasks.

2. We optimize the file system to better utilize the benefits of flash in our software stack, while minimizing the cost of adding flash by selectively storing metadata on SSDs.

3. Experimental results using our techniques show orders of magnitude improvement in IOPS and reduced latencies in nonsequential workloads. Even traditional backup workloads show 25%-30% improvement in restore throughput because the SSD cache reduces disk accesses. In experiments where both traditional and NSIO workloads execute concurrently, our system maintains high performance for both workloads.

The rest of the paper is organized as follows. §2 provides a brief overview of deduplication in file systems and recent changes in backup applications that motivated us to re-architect our file system design. §3 describes our high-level architecture and design and §4 presents the detailed file system modifications in the DDFS stack. §5 states our experimental platform and workloads used in our study. Detailed experimental results are provided in §6. We discuss related work in §7. §8 concludes our study and discusses future extensions.

2 Background and Motivation

Here we provide an overview of our protection storage and a more detailed discussion of the changes that motivated our architecture modifications.

2.1 Deduplicating Protection Storage

Deduplication is common in commercial products and has been widely discussed including survey articles [30, 35]. Here we provide a brief overview of the specifics of our system; see Zhu, et al. [36] for additional details.

Each file is represented by a Merkle tree [26], which is a hierarchical set of hashes. The lowest level of the tree is file data, and hashes to many chunks[2] are aggregated into a new chunk one level higher in the tree. The fingerprint of that chunk is stored a level higher, and so on. The root of the Merkle tree represents a recipe for a single

[1] The data cache is not yet commercially available.

[2] In the interest of brevity, we shall refer to the unit of deduplication as a chunk even if the system uses fixed-sized blocks.

file, and the filename-to-recipe mapping is managed by a *directory manager*. Chunks are aggregated into write units, called *containers*, which are megabytes in size, and may be compressed in smaller units of tens of chunks.

Thus if a file is read sequentially, the system uses the directory manager to find the root of the Merkle tree, makes its way to the lowest level of metadata, and identifies the fingerprints of potentially thousands of data chunks to access. The chunks themselves may be scattered throughout the system, though there are techniques to alleviate read fragmentation [13,20]. There is an index that maps each fingerprint to a container, and the container has metadata identifying chunk offsets.

The storage system is log-structured [32], so whenever new data or metadata chunks get written, they are added to new containers. Garbage collection is necessary to reclaim free space, which can arise from deleted files removing the last reference to a chunk, as well as extra duplicates that are written due to imperfect deduplication [11].

2.2 Changing Environments

The improvements to our deduplicating storage system were motivated by changes in hardware and in applications [3]. Dramatic increases in the capacity of individual disks meant that a purely disk-based system would not have sufficient input/output operations per second (IOPS) to perform the necessary index lookups to deduplicate effectively. Moving the index to SSDs was a necessary step to improving performance, but it was not sufficient to handle the other changes.

The most extreme requirements on performance derive from two changes in workload. Initially, our systems had to deal with the change from periodic full backups to generating full backups by transferring changes since the last backup with *virtual synthetic* full backups [3,12] and change-block tracking [33] backups. With virtual synthetic and change-block tracking, changes to a backup would be written into protection storage, then a new "full backup" would be created by making a copy of the file metadata with the changes included. This would often be done at intervals of hours, rather than the weekly periodic backups from traditional workloads; thus the amount of metadata in the file system would grow by orders of magnitude, could not be cached in DRAM, and was slow to access on disk. The access patterns also changed from fully sequential writes (full backups, which would write a backup from start to finish on a regular basis) to "incremental" writes that would have monotonically increasing offsets but might skip large regions of a file.

Even greater stress to system performance arose with scenarios where data in protection storage are accessed *in place* after a failure. As an example, a virtual machine (VM) backed by a `vmdk` file might be booted and run from protection storage even while its `vmdk` image is migrated to a primary storage server. Accesses to any data not yet received by primary storage would be served through explicit I/Os from protection storage. It is characterized by nonsequential accesses, with additional sequential accesses introduced by storage migration in the background. Read operations during this period often include access of backup data for browsing and recovery of small files from large backups. Read-write operations from a running VM further stress deduplicating storage due to nonsequential reads and overwrites. This is somewhat analogous to storage vMotion [24], when a virtual disk can be accessed while it migrates. This workload in turn has implications on data formats, deduplication units, and physical devices.

Data formats A number of data protection applications perform transformations on data during the backup process. For example, some legacy backup applications have been described as creating a *tar*-like file concatenating data and metadata from many individual files in primary storage, to create a single backup file in protection storage [22]. It is not feasible to restore an individual file from this large aggregate, so there has been a shift towards backing up individual files in their "native" format. This in turn can lead to millions or billions of individual files, making the performance of namespace operations very important.

Unit of deduplication While content-defined chunking is a well studied topic, there is usually an assumption of little knowledge about the data type. As an example, without application-specific knowledge, variable-sized chunks are generally able to localize the impact of small edits when forming chunks. When application knowledge is available, it can increase efficiency such as deduplicating virtual machine disk images [14] in fixed-size units corresponding to disk blocks. (That is, updates to one part of the file do not shift content in other parts of the file.) More generally, application-specific deduplication must align the unit of deduplication appropriately, whether it is a database record or a block storage system directly performing backups.

Devices In addition to using SSDs to store metadata such as the deduplication index, we need to cache file metadata (the recipes that uniquely identify the individual chunks within a file) and file data blocks themselves. SSD caching, including the implications of retrofitting this to an existing disk-based data protection system, is a focus of this paper.

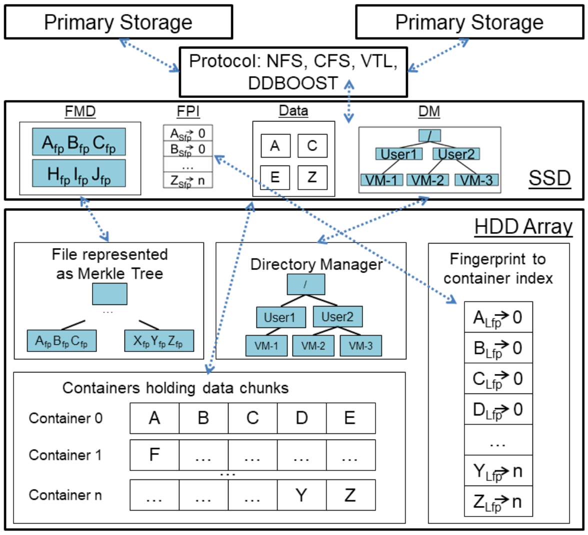


Figure 1: SSD caches: file metadata (FMD), fingerprint index (FPI), data, and directory manager (DM)

Mixed workloads While DDFS was originally designed to support large sequential access patterns, the shift to new workloads does not mean systems no longer have sequential accesses. Instead, there can be mixes of both across and within files. As an example, a system may restore a VM image by sequentially accessing it to create a copy on another system, but during the restore operation the VM image is accessed with reads and writes at more arbitrary locations. There may also be accesses related to ongoing backups relative to this file. DDFS needs to treat the types of accesses differently (*e.g.*, prefetching file data and metadata for the sequential restore) and provide different qualities of service based on resource requirements.

3 Modernizing Protection Storage

To motivate our caching decisions, consider the steps necessary to access data within a file at an arbitrary offset. Figure 1 shows four caches: file metadata (FMD), fingerprint to container index (FPI), data, and directory manager (DM). They are shown as SSD caches, though initially they existed only in DRAM. First, we find the entries in the file's Merkle tree corresponding to the desired data offset (FMD cache). Traversing the tree itself involves a level of indirection as every chunk within the tree is referenced by hash which is translated to a container using a fingerprint index (FPI cache). We then read in the portion of the file tree, which leads us back through the fingerprint index to access data chunks (data cache) that are returned to the client. Please note that the fingerprint index is shown in a simplified form relative to updates discussed in §4.7. Finding the file's top-level information involves a directory structure (DM cache) that is also used for namespace changes.

For largely sequential accesses, the overhead of retrieving various types of metadata will be amortized across many data accesses. For instance, if an application reads 1 MB of fixed-sized 4 KB chunks (256 in total), and the fingerprints of those chunks are all contained in a single chunk in the Merkle tree, then the cost of the directory lookup and the first few levels of the Merkle tree are amortized across 256 chunk reads. If the locality of those chunks is high, the first lookup in the FPI will lead to a container that populates the DRAM FPI for the rest of the 1-MB read.

For random accesses, especially to individual files, each read can result in a DM lookup, on-disk Merkle tree traversals, on-disk FPI lookups, and finally a data access. While caching will not significantly help completely random accesses, any amount of locality can result in substantial improvement.

We therefore use SSD to cache several metadata structures as well as file data. We controlled the costs of our design by using low-cost SSD that totaled 1% of the total hard drive capacity. Our selected SSDs only support three full erasures per day, so our design attempts to minimize writes. At the time of writing this article, SSD costs approximately 8× more per GB than HDD, so adding a 1% SSD cache increases the hardware capacity costs by 8% [1]. While some backup customers appreciate all-flash options, many remain sensitive to costs.

3.1 Caching the File Metadata

While our system attempts to group metadata chunks together, locality can become fragmented for multiple reasons, such as GC repositioning chunks and related files sharing previously written chunks. To decrease the latency for accessing file metadata (FMD) (i.e. the Merkle trees), we cache FMD in flash. Also for NSIO, accessing each data chunk requires accessing a metadata chunk that is unlikely to be reaccessed in the near future. This doubles the number of I/Os needed to serve a client request, so prefetching metadata chunks and caching in flash will decrease overall latency. There are multiple challenges we considered while designing the FMD cache.

We noted that metadata chunks can be of variable size, and not align with a flash erasure unit. We therefore packed metadata chunks into a multi-MB cache blocks and created a caching policy similar to Nitro [18]. Briefly, we maintain a single time stamp per cache block and perform LRU eviction using that time stamp to evict an entire cache block at a time. While LRU is a simple policy, and more advanced techniques [19] could be used, we have found that chunks within cache blocks tend to age at similar rates. We track the amount of data written to the FMD and will throttle new insertions to maintain our long term average of three writes per day times the capacity of the cache.

Importantly, we must determine which metadata chunks to add to the cache, as our capacity is insufficient to cache metadata chunks for all files in the system. 10% of the SSD cache is allocated to the FMD. Rather than simply inserting all FMD, we use admission control to determine what is appropriate to cache (§4).

3.2 Caching the Fingerprint Index

For any given chunk, on the read path the system must map from its unique fingerprint to its location in storage. The fingerprint to container index (FPI) performs that function. Historically, in DDFS the index would be on disk, with a small subset cached in DRAM. Our index design actually requires two I/Os for each access because there are typically two layers to the index. On a read operation, where the FPI mapping was not in the cache, there would be I/O to disk. However, the container that would then be loaded would include metadata for other chunks in the container, and their fingerprints would be cached. Since reads were typically large restore operations, accesses would be sequential and many other fingerprints would be found in the cache. As lower cost, denser hard drives have become available, they have been added to our systems. Unfortunately, IOPS per capacity have decreased for denser drives, and this further motivates the need to use SSD to accelerate NSIO such as fingerprint index accesses.

To support NSIO, the system keeps the entire FPI in SSD, but because space is limited, DDFS makes a concession. Each record stores a short version of each fingerprint in SSD along with the corresponding container and other information. Rather than storing all 20 bytes of a fingerprint, it stores four bytes. In the case of duplicate short fingerprints, only one copy is recorded in the FPI. More details are in §4. Figure 1 labels the full fingerprints as Lfp on disk and the short fingerprints as Sfp in SSD. It is possible the FPI will incorrectly match a query fingerprint based on the first bytes of a short fingerprint in the cache, but this false positive case will be detected. If the needed chunk is not found in the container referenced by the short fingerprint, then the full on-disk index is consulted. Latency is higher in this case, but as it is infrequent, overall performance improves dramatically while controlling SSD costs. FPI occupies 50% of the SSD cache.

For the data locality of traditional backups, for every 1MB external read, we issue an average of eight I/Os to disk where two are for the FPI. When the FPI is moved to SSD, we should see a benefit of at least 25% on disk bound systems. For data with bad locality, we will issue multiple FPI lookups per client read, so FPI in SSD would offer even more benefit. In Figure 2, we compare overall throughput of our system when the FPI is in SSD versus only on HDD. We show that having a fingerprint cache in SSD improves restore performance at higher stream counts, when disk is a bottleneck, by up to 32%. More experimental details are provided in §6.

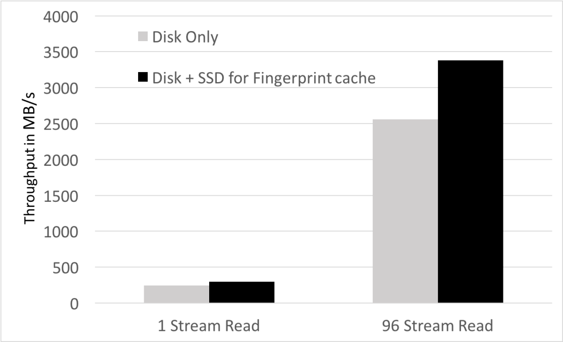


Figure 2: Performance evaluation of a NSIO workload with and without the fingerprint index cache in SSD.

3.3 Caching the Chunks

Once the system knows where to find a chunk, it loads the storage container holding it. With traditional workloads and significant spatial locality, two properties hold that are not true for NSIO workloads:

1. Once accessed, a particular chunk is unlikely to be accessed again unless the same content appears multiple times in the restore stream.

2. Other chunks in the same storage container are reasonably likely to be accessed as well, so the system benefits from caching that container's data and metadata. The container metadata can be used to avoid FPI lookups when locality is high [36].

For NSIO, in contrast, the locality of access within a container may be highly variable, and the reuse of specific data may be more commonplace. For instance, a data chunk might be written and then read, with a gap between the accesses that would be too large for the data to reside in a client or server DRAM cache. The SSD data cache is intended to provide a large caching level to optimize those access patterns, but it needs to dynamically identify what patterns it encounters.

On a data miss for sequential reads, we load the desired chunk as well as the following chunks that may be accessed. This helps to warm our cache and improves access times. We avoid loading chunks during writes except when read-modify-write operations are necessary. Techniques to identify such cases and modifications to DDFS are described in §4. 35% of the SSD cache is reserved for data chunks.

3.4 Caching Directories

The directory manager (DM) manages the mapping from file paths to Merkle trees. Thus, the first time a file is opened, the system must access this mapping to find the root of the tree. For large files such as VM images that are opened once and then accessed over time, the cost of the DM lookup is insignificant (once for a large file), but if there are numerous files to open (such as the result of backing up a file system as individual files); this cost can be a significant performance penalty.

In DDFS, data for DM resides on HDD, but a full copy is now cached in SSD for performance. Since such a cache is straightforward, our experiments focus on applications that are not namespace-intensive, so we do not consider the DM cache further. DM is allocated 5% of the SSD cache.

4 File System Modifications to Support Nonsequential Workloads

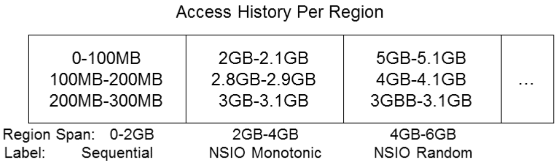
Our goal is to enable faster accesses for new workloads while continuing to support traditional sequential backup/restore workloads without performance degradation. Besides the flash caches described previously, numerous changes were needed in the file system to support NSIO. We begin by presenting our technique for identifying the type of client access, which determines if optimizations are applied. We then describe the most important file system changes.

4.1 Detecting Workload Types

To decide whether NSIO processing is needed, the incoming I/O requests must be analyzed to determine the type of access. Defining “sequential” is itself a challenge, as access patterns may not be strictly *sequential* even if they are *predictable* [17].

The access pattern detection algorithm partitions large files into regions and keeps a history of recent I/Os (specifically, 16 I/Os) per region as shown in Figure 3. There are two kinds of detection to check for data sequentiality and access patterns.

By default, all incoming I/Os are assumed to be sequential until there is enough history of previous I/Os to check for other types. Once the history buffer is full, if a new I/O is not within a threshold distance of one of the previous 16 I/Os, it is considered nonsequential.



Access History Per Region

	0-100MB 100MB-200MB 200MB-300MB	2GB-2.1GB 2.8GB-2.9GB 3GB-3.1GB	5GB-5.1GB 4GB-4.1GB 3GBB-3.1GB	...
Region Span:	0-2GB	2GB-4GB	4GB-6GB	
Label:	Sequential	NSIO Monotonic	NSIO Random	

Figure 3: Access history for three regions of a file, labeled sequential, NSIO monotonic, and NSIO random.

The reason for comparing with several past accesses is to avoid detecting re-ordered asynchronous I/O operations from a client as NSIO. By keeping multiple regions of access patterns within a file, we allow combinations of sequential and NSIO accesses to the same file to coexist without NSIO patterns hiding the existence of simultaneous sequential access. One example of this is NSIO from accessing a live VM image while simultaneously performing vMotion; another is the result of reordering of asynchronous I/O operations on a client. Our region size is a minimum of 2GB and grows to maintain at most 16 regions per file. The memory required for tracking a file is ≤3KB.

Referring to Figure 3, besides sequential I/O, we also label two variants of NSIO: NSIO monotonic and NSIO random. Monotonic refers to accesses that are to the same or non-consecutive increasing offsets. Random refers to accesses that do not have a discernible pattern. The monotonic pattern is particularly common when a backup client generates a synthetic full backup by first copying the previous full backup (an efficient metadata operation in deduplicated storage) and then overwrites regions at increasing offsets in the file. Distinguishing NSIO monotonic from random patterns allows us to implement different caching and eviction methodologies.

4.2 Prefetching Content

One of the uses of identifying accesses based on history per region is to prefetch and cache content. Importantly, we also avoid caching content that will not be reused. Our options are to load data into DRAM for immediate use or load into SSD if reuse is expected.

Specifically, when access patterns are labeled as sequential or NSIO monotonic, we can prefetch and load into DRAM, because we know the data or metadata will be used soon and SSD caching is unlikely to provide further benefit. For NSIO random I/O, we prefetch into SSD because we need to cache most of the active data set, which is larger than DRAM, to get the benefit of caching. In order to warm-up the cache sooner for every file with NSIO random I/O, we first load 128KB around the current I/O (e.g. 8KB) for caching in SSD since it may be reused, and there is little additional latency for

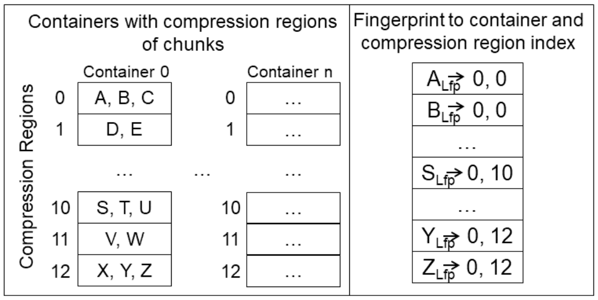


Figure 4: The fingerprint index is updated to map from fingerprint to container and compression region. The structured is presented in a simplified form.

loads of that size in our system.

4.3 Direct Read of Compression Regions

As described in §3, accessing a region of a file involves identifying the fingerprint representing that chunk from the file recipe, checking the fingerprint index for the corresponding location on disk, and then reading the chunk. For sequential accesses, we implemented an optimization in the early versions of the file system. The fingerprint index maps to container, so we read in the container's metadata region into RAM, which consists of a list of fingerprints for chunks within each compression region. We then determine which compression region to read and decompress to find the needed chunk. For sequential (or nearly sequential) accesses, we typically find most needed fingerprints in the RAM cache without the need to query the fingerprint index [36].

While this previous optimization dramatically reduces fingerprint index accesses for sequential I/O, it is inefficient for NSIO. A client's nonsequential read requires a fingerprint index read, container metadata read, and compression region read, *i.e.* three reads in total. Because future accesses are unlikely to remain within the same container, there is no amortization of reading a container's metadata. To remove the container metadata read for NSIO cases, we adjusted our fingerprint index to map from fingerprint to a compression region offset and size *within* a container (Figure 4). This allows us to perform direct compression region reads without first reading in container metadata, reducing the number of accesses from three to two. We dynamically decide based on access patterns whether to read compression region metadata or not.

To reduce SSD space for the FPI entries, we limit the entry size to twelve bytes. Four bytes come from the shortened fingerprint. Four bytes are used for the container ID, which is sufficient since it is relative to the lowest container ID within the system. The remaining four bytes are used to describe the compression region within the container with bits allocated to the compression region offset and size within the container as well as internal uses. To reduce the number of bits required, compression regions are written at sector boundaries. When indexing a fingerprint, we use a hash of the first eight bytes to select a FPI bucket. In combination with the four bytes short fingerprint, the collision rate is below 0.01%.

4.4 Higher Concurrency with Queue Changes

Applications directly accessing files from backup storage have high performance requirements, and latency is an important aspect, so I/Os must be processed as soon as they enter the file system. Unlike traditional workloads that tend to be highly sequential, with one client I/O effectively dependent on earlier I/Os to complete, NSIO has a greater need and opportunity for parallelism. For NSIO, FMD required to process the I/O may not be in memory and will require disk I/Os. Requests that are dependent on the same FMD will be processed serially in the order received; however, requests that do not require the same FMD are processed in any order and in parallel. Once the required FMD is loaded for any I/O, that request is given priority for further processing to avoid starvation. Apart from issuing parallel I/Os for FMD on disk, fingerprint lookups for multiple reads within the same file take place in parallel for NSIO.

4.5 Adjusting the Chunk Size to Improve Nonsequential Writes

For traditional large backup files, variable chunking achieves better deduplication than fixed-size chunks because it better identifies consistent chunks in the presence of insertions and deletions [34, 36], which we refer to as *shifts*. For new use cases that have block-aligned writes, such as change block tracking for VMs, shifts do not occur, and fixed-sized deduplication is effective [14]. Although variable-sized chunking has better deduplication, the performance gains achieved with fixed-size chunks outweighs the deduplication loss [27].

Based on customer configuration or backup software integration, we label workloads that will benefit from fixed-sized chunks. This simplifies the write path, as we do not need to find chunk boundaries or perform a read-modify-write for a partial overwrite of a chunk. For certain applications, such as VMs and databases, the block size is predetermined, and we set our chunk size accordingly for further efficiency.

4.6 Delayed Metadata Update

Switching to fixed-sized chunks has the added benefit that it allows for more efficient updates of file recipes during nonsequential writes. Traditionally, we would need to read in portions of the file recipe to provide fingerprints for chunks that must be read before being modified. With fixed-size chunks, we never need to modify existing chunks as they are simply replaced. We do need to update the file recipe to reference new chunks, but this can be delayed until sufficient updates have been accumulated. Since our chunk references are 28 bytes, 1MB of non-volatile memory can buffer references for nearly 300MB worth of logical writes.

4.7 Selective Fingerprint Index Queries

While our file system is designed to perform deduplication by identifying redundant chunks, we may choose to skip redundancy checks to improve performance [3]. We have found that nonsequential writes tend to consist of unique content. So to avoid fingerprint index queries that are unlikely to find a match, we disable querying the fingerprint index for small nonsequential writes (<128KB). Any duplicate chunks written to storage will be removed during periodic garbage collection [11].

4.8 Quality of Service and Throttling

DDFS has a quality of service (QoS) mechanism that assigns shares for external and internal workloads such as backup, restore, replication and garbage collection. These shares are used in the CPU and disk scheduler to provide QoS for the workloads. NSIO can happen as part of backup or restore, so we made changes to further split the backup and restore workload shares into sequential and nonsequential shares. The number of shares assigned to these workloads is tunable based on a customer's desired system behavior. By default, the shares for NSIO workloads are kept at 20% so as to not impact other critical workloads, but as reads and writes on backups during a restore becomes commonplace, shares may need to be increased for NSIO.

On non-uniform memory access architectures, jobs pertaining to a task are assigned a particular CPU for cache locality. Our earlier implementation used round robin assignment of jobs to CPUs. However, the resource requirements between NSIO workloads vary greatly and hence a simple round robin is insufficient. In the latest version of DDFS we have changed this assignment to least-loaded CPU instead. NSIO performance greatly depends on read performance, so we have modified our I/O scheduler to avoid read starvation and provide higher priority for read requests.

With all of the changes to increase NSIO performance, accepting more I/Os in parallel at the protocol layer usually improves overall performance. However, beyond a limit, further client requests will cause RPC timeouts, and hence I/O throttling per workload type becomes important. Based on the type of workload and the average latency, we have implemented an edge throttling mechanism where the protocol layer can query the subsystem health and insert queue delays to dynamically change the number of client accesses supported.

5 Experimental Methodology

This section describes our experimental methodology and the test environment including the system configuration and workloads used. All our results are measured on a Data Domain DD9800 [10] configured with maximum capacity. It has 60 Intel(R) Xeon(R) CPU E7-4880 v2 processors @2.50GHz, with 775GB DRAM, 8 10Gb network ports, 10.9TB SSD, and 1008 TB disk storage across 6 shelves with 4TB HDDs. Each shelf has between 1 and 4 packs, with 15 HDDs per pack. There are 20 spare HDDs. We produce accesses to the DD9800 using up to 8 clients running Linux version 2.6.32 with Intel(R) Xeon(R) CPU E5-2620 with 2.00GHz cores, 64 GB of memory, and a 10Gb Ethernet card.

We primarily use traditional and NSIO workloads for our measurements. Performance numbers for traditional backup and restore are reported using an in-house synthetic generator that randomly creates first generation backups for each stream and then modifies following generations with deletions (1%), shuffles (1%), and additions (1%) [7]. Across clients, the total size of first generation backups is 3TB, and metadata is approximately 1% of the data size. We wrote every 5th generation, though we allowed changes to accumulate in memory even for unwritten generations. This workload has 100% sequential read/write accesses to data for all generations. However, metadata accesses are NSIO. We report throughput numbers as the average of generations 41 and 42.

For a NSIO workload, we use the industry standard FIO benchmark [6] to simulate large sequential and NSIO reads as well as small NSIO reads and writes. We also present results when accessing 32 100GB VM images with mixtures of sequential I/O and NSIO, as described in each experiment. While customer VMs often share content, in order to reduce factors affecting our experiments, we have confirmed that there was no potential deduplication within or across the images. Here we report performance numbers in terms of IOPS and average latency. Unless otherwise noted, experiments were performed on an isolated system configured with fixed-sized chunks and without other read/write operations or background tasks such as garbage collection or replication.

All metadata fit within SSD without the need for admission control.

We show the benefits of our hardware and software optimizations in the following experiments. Each data point in our experiments with traditional backup workload was collected in runs that lasted multiple days and on data sets that were aged up to 42 generations of backups. Multiple clients were used to generate the backup workload, and results are averaged across clients. Each data point in our experiments with NSIO workload was collected by measuring the average performance (IOPS and latency) with at least three runs, and the standard deviation of results is <1.5% in all cases.

6 Evaluation

We begin by exploring the impact of caching metadata and data as well as software optimizations within DDFS for NSIO workloads. Then we investigate the impact on traditional, sequential workloads using different protocols. Finally, we study the sensitivity to different read/write ratios in NSIO workloads and the impact when storage vMotion occurs in parallel.

6.1 Caching and Software Optimizations

We investigate the impact of progressively adding metadata and data to a SSD cache as well the value of software optimizations in terms of average IOPS (Figure 5(a)) and latency (Figure 5(b)). Metadata include the FPI, FMD, and DM caches, though our tests do not perform directory operations. To avoid direct comparisons, the experiments with optimizations disabled are separated by a dashed line in each set of bars

We vary the number of VM images accessed from 1 to 32 and plot the average IOPS and latency for a NSIO workload. In these experiments, we study a read-only workload, and each VM is issued a maximum of 8 concurrent I/Os. When the flash cache is disabled, each external I/O will translate to six internal I/Os to disk. This includes two I/Os for FPI lookup to then perform one I/O for file metadata. From the file metadata, we have the chunk fingerprint and then perform two I/Os for FPI lookup and one I/O to load the data. The total number of HDD IOPS available on the test system is 24K. So, the theoretical achievable client IOPS when the cache is not available would be 4,000. Software optimizations are enabled except in one set of runs.

We see in the experiment with 32 VMs and the cache disabled, we achieve 3,200 IOPS with a latency of 35ms. When we enable the caching of metadata, every external NSIO will result in one I/O to HDD for data. We show that we can achieve 28K IOPS in a 32 VM experiment, with an average latency of 10ms. When both data and metadata are in the flash cache, IOPS are only limited by the data set size we can cache. On the test system, we can cache 100% of the data for up to 24 VMs and 75% of the data with 32 VMs. We achieve peak performance of 57K IOPS for 24 VMs with a cache hit ratio of 95%. The overall latency stays under 5ms even at peak IOPS.

We next consider the benefit of software optimizations (§4) to improve NSIO performance. Results show that using a SSD cache for NSIO without software changes would limit us to a peak NSIO performance of 20K IOPS, compared to 56K IOPS when software enhancements are enabled. Similarly, even when data and metadata are cached, latency decreases from 13ms to 5ms with the addition of software optimizations.

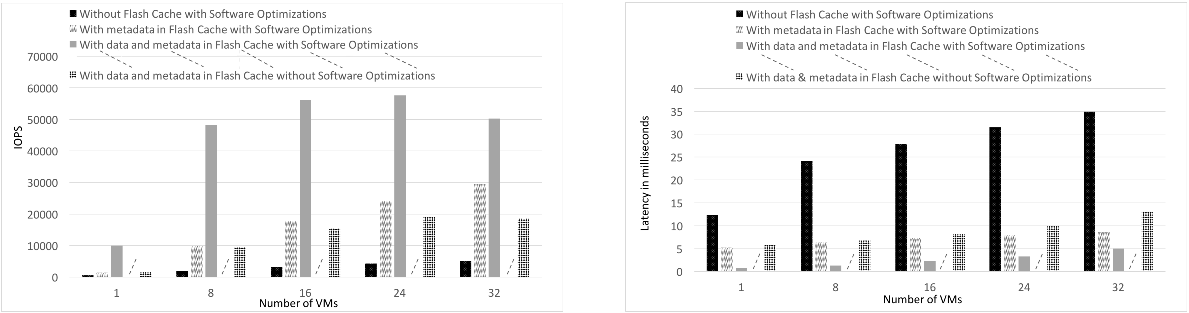
With a small cache and high churn in application workload, some portion of the I/Os will be serviced from disk. Our software optimizations remove unnecessary I/Os to disk (§4.2, §4.3, and §4.5), increase parallelism (§4.3), and improve the I/O scheduler (§4.8). With these changes, we are able to to achieve high NSIO IOPS and maintain a low latency with a SSD cache sized at 1% of the total system capacity.

6.2 Traditional and NSIO Workloads

In this experiment, we evaluate both traditional and NSIO workloads running concurrently to measure the impact on traditional workloads. We run both workloads through NFS and DDBOOST protocols. DDBOOST is our proprietary protocol where segmenting and fingerprinting of data is offloaded to backup clients and only changed data is sent across the network [12]. DDBOOST performance is typically higher than NFS because deduplication reduces the amount of data transferred, and the backup server has fewer computational demands.

In this experiment, we throttle NSIO workloads on 32 VMs to a total of 10K IOPS. 2.4TB of the 3.2TB data set fit in the data cache. In Figure 6, we measure the performance of 96 streams of backup and restore workloads while varying the protocol and the fraction of reads versus writes of the NSIO workload. A 100% read NSIO workload is possible when the client writes are redirected to primary storage during a recovery operation. 70% reads are common in other recovery use cases where both writes and reads are directed to backup storage. Read/write numbers represent restore and backup performance for high-generation backups with an equal split of 48 backups and 48 restores.

Considering the difference between NFS and DDBOOST, we find the expected result that DDBOOST has higher overall throughput because of offloading tasks to clients. Across protocols, backup and restore performance is not degraded more than than 10% when NSIO runs in parallel, though there is greater impact



(a) IOPS

(b) Latency

Figure 5: Average IOPS and latency as caching and software optimizations are varied. Up to 24 VMs, 100% of the data can be cached. Caching decreases to 75% for 32 VMs. Experiments without software optimizations are separated from the rest by a dashed line.

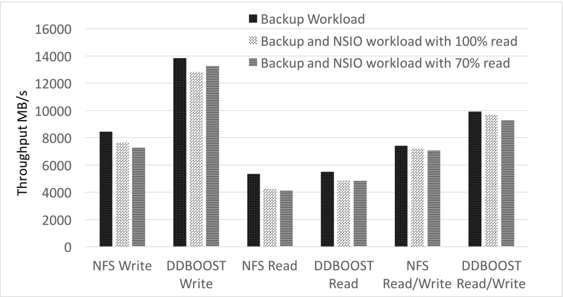


Figure 6: Traditional backup workloads have a 10% degradation when NSIO workloads are added with the DDBOOST protocol outperforming NFS.

when NSIO includes writes.

When more NSIO performance is required than the sustained IOPS specified for the product, a system level QoS parameter (§4.8) allows users to choose the amount of impact on traditional workloads they find acceptable to further increase NSIO performance. Though not shown due to space limitations, we experimented with varying the QoS share allotted to NSIO versus sequential workloads. As the share for NSIO increased from 25% to 50%, IOPS increased by 32%. Increasing the share from 50% to 75% increased IOPS 18% more. When NSIO was allocated 100% of the resources, IOPS increased an additional 125% due to the complete removal of sequential I/O interference.

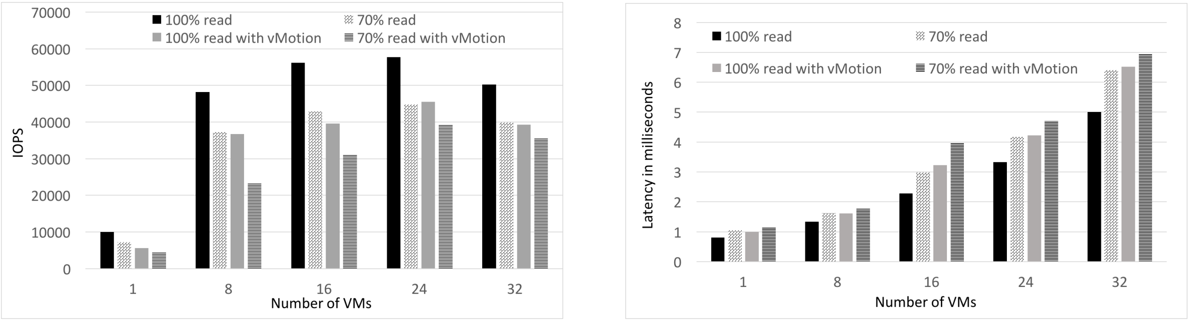
6.3 Performance during Restores

Some backup applications provide a Instant Access/Instant Restore feature where an application may be able to perform read/writes from the backup copy while a restore takes place. This feature may expose a read-only copy of a backup image for the applications to access while redirecting any writes to a write log typically located on primary storage. We simulate this workload using 100% NSIO reads. Other backup applications expose a read/write copy and send both reads and writes from the application to the exposed copy. This is simulated using a 70/30% reads/write NSIO workload. While a VM image is being accessed, backup applications also offer an option to perform storage vMotion of the VM back to primary storage. This workload is simulated by issuing sequential reads on the same VM image on which NSIO is taking place.

Figures 7(a) and 7(b) show an experiment where NSIO activity takes place with either 100% reads or 70/30% read/writes. With 24 VMs we see a peak of 56K IOPS and under 4 ms of latency with 100% reads. For the 70/30% read/write mix, we see a peak of 44k IOPS at 24 VMs where the cache gets nearly 95% hits. We also show that when vMotion on the same VM takes place, IOPS for NSIO drop by at most 20% and achieves a peak of 45K IOPs for 100% reads. At 70/30% read/write with vMotion, we achieve 40K IOPS. The overall result is acceptably high performance NSIO performance while vMotion takes place.

6.4 Fingerprint Cache Impact on Backup and Restore Workloads

For traditional backup workloads, even with software optimizations, disk I/O becomes a bottleneck. We previously presented experimental results for placing a FPI cache in SSD in Figure 2. For this test, we limited the total IOPS available on the test system to 4,200 by using only four disk groups, the smallest configuration possible. Other metadata and data accesses may still go to HDD, so the overall throughput improvement has many components besides fingerprint access speed. With a high stream count of 96, overall throughput increases with the SSD cache by up to 32%, which corresponds to the fraction of I/O that can be satisfied by the FPI cache in SSD.



(a) IOPS

(b) Latency

Figure 7: Average IOPS and latency for NSIO read/write access to VMs, with and without vMotion in parallel.

7 Related Work

A recent article [3] described the changes to data protection workloads and some of the changes DDFS made to address them. However, the changes were described very generally, focusing on qualitative issues but not quantitative ones. As an example, the article mentioned the addition of SSDs, short fingerprints in SSD, and selectively writing duplicates, but there were few implementation details and no experiments. While we have focused on improvements to backup storage for NSIO, backup software drives most of the client-initiated workloads [4, 8]. The work on improving garbage collection enumeration performance [11] to handle high deduplication rates and numerous individual files provided detailed performance measurements, but that effort is largely orthogonal to the improvements for NSIO described here.

SSD-assisted deduplication has taken many forms. DedupeV1 [25] and ChunkStash [9] were two early systems that moved the fingerprint index into SSD to improve performance. ChunkStash used Cuckoo Hashing [29] to reduce the impact of hash collisions, something we have not found to be a significant performance issue. PLC-cache [23] categorized deduplicated chunks by popularity to determine what to cache. Nitro [18] presented a technique for caching and evicting data chunks in large units to SSD to improve performance while reducing SSD writes, which influenced our metadata and data cache design. Kim et al. [16] modeled deduplication overheads and benefits within SSD and then accelerated performance with selective deduplication against recently written fingerprints. We view the contribution of our work as lessons learned from a deployed storage system pertaining to caching, prefetching, and scheduling, and not simply the addition of SSDs.

While there have been multiple papers regarding sequential write and read performance for deployed deduplicated storage products [5, 15, 20, 21], there has been little discussion of nonsequential workloads. Discussing the architectural changes needed to support both sequential and NSIO workloads in deduplicated storage will hopefully drive further research.

8 Conclusion and Future Work

New workloads for backup appliances and denser HDDs have placed demands on backup storage systems. DDFS has had to evolve to support not only traditional workloads (full and incremental backups with occasional restores) but also newer nonsequential workloads for thousands of customer deployments. Such workloads include direct access for reads and writes in place, as well as other workload changes such as storing individual files and eschewing periodic full backups. Additionally, traditional and newer workloads must *peacefully coexist* within the same product.

Because of the cost difference between SSDs and disk, we have chosen to cache a limited amount of metadata and file data in SSD rather than moving the entire system to SSD. We demonstrate that these caches not only improve NSIO by up to two orders of magnitude, but our system can also simultaneously support traditional workloads with consistent performance. In summary, improvements to our software and the addition of SSD caches allow DDFS to support both new and traditional workloads.

In the future, we expect NSIO workloads to become more common as customers increase the frequency of backups. In combination with decreasing SSD prices (though likely still more expensive than HDD), it may become worthwhile to increase our SSD cache to include most metadata and a larger fraction of active data. We will need to revisit our software design as bottlenecks shift between I/O and CPU.

Acknowledgments

We would like to acknowledge the contributions of the DDFS team including: Uday Kiran Jonnala, Sirisha Kaipa, Vrushali Kulkarni, Shuang Liang, Valiveti Narasimha, Shantanu Patwardhan, Balaji Subramanian, Pradeep Thomas, Satish Vishwanathan, Grant Wallace, Sailu Yallapragada, and Sean Ye. We appreciate the feedback of our shepherd Xiaosong Ma and the anonymous reviewers.

References

[1] Pc part picker price trends. `https://pcpartpicker.com/trends/price/internal-hard-drive/#storage.7200.6000000`. Accessed: 5-7-2018.

[2] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 1–14.

[3] ALLU, Y., DOUGLIS, F., KAMAT, M., SHILANE, P., PATTERSON, H., AND ZHU, B. Backup to the future: How workload and hardware changes continually redefine Data Domain file systems. *Computer 50*, 7 (2017), 64–72.

[4] AMVROSIADIS, G., AND BHADKAMKAR, M. Identifying trends in enterprise data protection systems. In *USENIX Annual Technical Conference (ATC'15)* (July 2015).

[5] ARONOVICH, L., ASHER, R., BACHMAT, E., BITNER, H., HIRSCH, M., AND KLEIN, S. T. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009).

[6] AXBOE, J. FIO. `http://git.kernel.dk/fio.git`, 2018. Retrieved Feb 1, 2018.

[7] BOTELHO, F. C., SHILANE, P., GARG, N., AND HSU, W. Memory efficient sanitization of a deduplicated storage system. In *USENIX Conference on File and Storage Technologies (FAST'13)* (Feb 2013).

[8] CHERVENAK, A., VELLANKI, V., AND KURMAS, Z. Protecting file systems: A survey of backup techniques. In *Joint NASA and IEEE Mass Storage Conference* (1998), vol. 99.

[9] DEBNATH, B. K., SENGUPTA, S., AND LI, J. Chunkstash: Speeding up inline storage deduplication using flash memory. In *USENIX Annual Technical Conference (ATC'10)* (2010).

[10] DELL EMC. Data domain dd9800 specification sheet. `http://www.emc.com/collateral/specification-sheet/h11340-datadomain-ss.pdf`, 2017. Retrieved Feb 1, 2018.

[11] DOUGLIS, F., DUGGAL, A., SHILANE, P., WONG, T., YAN, S., AND BOTELHO, F. C. The logic of physical garbage collection in deduplicating storage. In *15th USENIX Conference on File and Storage Technologies (FAST'17)* (2017).

[12] DOUGLIS, F., HUBER, A., LEWIS, D., AND TRAYLOR, R. Experiences with a distributed deduplication API. In *Massive Storage Systems and Technologies (MSST'17)* (2017), IEEE.

[13] FU, M., FENG, D., HUA, Y., HE, X., CHEN, Z., XIA, W., HUANG, F., AND LIU, Q. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *USENIX Annual Technical Conference (ATC'14)* (2014).

[14] JIN, K., AND MILLER, E. L. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), SYSTOR'09, ACM.

[15] KACZMARSKI, M., JIANG, T., AND PEASE, D. A. Beyond backup toward storage management. *IBM Systems Journal 42*, 2 (2003), 322–337.

[16] KIM, J., LEE, C., LEE, S., SON, I., CHOI, J., YOON, S., LEE, H.-U., KANG, S., WON, Y., AND CHA, J. Deduplication in ssds: Model and quantitative analysis. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on* (2012), IEEE.

[17] LI, C., SHILANE, P., DOUGLIS, F., SAWYER, D., AND SHIM, H. Assert(!Defined(Sequential I/O)). In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2014).

[18] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: a capacity-optimized ssd cache for primary storage. In *USENIX Annual Technical Conference* (June 2014), pp. 501–512.

[19] LI, C., SHILANE, P., DOUGLIS, F., AND WALLACE, G. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th International Middleware Conference (Middleware'15)* (Dec. 2015).

[20] LILLIBRIDGE, M., ESHGHI, K., AND BHAGWAT, D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *USENIX Conference on File and Storage Technologies (FAST'13)* (Feb 2013).

[21] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *USENIX Conference on File and Storage Technologies (FAST'09)* (2009).

[22] LIN, X., DOUGLIS, F., LI, J., LI, X., RICCI, R., SMALDONE, S., AND WALLACE, G. Metadata considered harmful... to deduplication. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2015).

[23] LIU, J., CHAI, Y., QIN, X., AND XIAO, Y. Plc-cache: Endurable ssd cache for deduplication-based primary storage. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on* (2014), IEEE.

[24] MASHTIZADEH, A., CELEBI, E., GARFINKEL, T., AND CAI, M. The design and evolution of live storage migration in VMware ESX. In *Usenix Annual Technical Conference (ATC)* (2011), vol. 11, pp. 1–14.

[25] MEISTER, D., AND BRINKMANN, A. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (2010), IEEE, pp. 1–6.

[26] MERKLE, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology CRYPTO'87* (1988), Springer.

[27] MEYER, D., AND BOLOSKY, W. A study of practical deduplication. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2011), pp. 229–241.

[28] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *18th ACM Symposium on Operating Systems Principles* (2001), SOSP'01.

[29] PAGH, R., AND RODLER, F. F. Cuckoo hashing. In *European Symposium on Algorithms* (2001), Springer, pp. 121–133.

[30] PAULO, J., AND PEREIRA, J. A survey and classification of storage deduplication systems. *ACM Computing Surveys 47*, 1 (2014).

[31] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02.

[32] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992).

[33] VMWARE INC. VMware vSphere Storage APIs – Data Protection (formerly known as VMware vStorage APIs for Data Protection or VADP) FAQ (1021175), Oct. 2016. `https://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1021175`.

[34] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *USENIX Conference on File and Storage Technologies (FAST'12)* (2012).

[35] XIA, W., JIANG, H., FENG, D., DOUGLIS, F., SHILANE, P., HUA, Y., FU, M., ZHANG, Y., AND ZHOU, Y. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE 104*, 9 (Sept. 2016).

[36] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *USENIX Conference on File and Storage Technologies (FAST'08)* (Feb 2008).

STMS: Improving MPTCP Throughput Under Heterogeneous Networks

Hang Shi
Tsinghua University

Yong Cui [*]
Tsinghua University

Xin Wang
Stony Brook University

Yuming Hu
Tsinghua University

Minglong Dai
Tsinghua University

Fanzhao Wang
Huawei Technologies

Kai Zheng
Huawei Technologies

Abstract

Using multiple interfaces on mobile devices to get high throughput is promising to improve the user experience. However, Multipath TCP (MPTCP), the de-facto standardized solution, suffers when different paths have heterogeneous quality. This problem is especially severe when the difference is the path latency. Our experimental results show that it causes the burst sending of packets from the fast path, which requires the in-network buffer to be big to achieve the full benefit of the bandwidth aggregation. In addition, it also requires bigger host buffer to fully utilize the fast path. To solve these problems, we propose and implement a new scheduler, which pre-allocates packets to send over the fast path for in-order arrival. Instead of relying on the estimation of network path condition, our scheduler dynamically adapts the MPTCP-level send window based on the packets acknowledged. Our evaluation shows that our scheduler can improve the throughput by 30% when the in-network buffer is limited, 15% when the host buffer is limited.

1 Introduction

There is a huge demand on network bandwidth with the rapid growth of network users and applications, as well as the emergence of bandwidth hungry applications such as multimedia streaming, cloud computing and virtual reality. To obtain high network throughput, a lot of recent interests have been drawn to exploit multi-path transmissions and aggregate the bandwidth through Multi-path TCP (MPTCP) [12]. As an example application, many wireless devices have two network interfaces, one to the local-area WiFi network and another to the wide-area cellular network. As wireless bandwidth is limited, a data stream can go through both networks to increase the transmission rate.

MPTCP is expected to be backward-compatible with conventional TCP and work with existing network components such as middle-boxes [26]. For the practical deployment of MPTCP, it is designed to be transparent to both applications and middle-boxes. From the perspective of applications, a single standard TCP is seen, whereas lower in the stack, MPTCP splits the data over multiple sub-flows. From the perspective of middle-boxes, each sub-flow is a normal TCP connection. MPTCP has already been implemented in Linux kernel [7] and used in iOS [4] and Giga Path in Korean Telecom [27]. Agache *et al.* [2] deploy MPTCP in the datacenter network to obtain better network utilization. Han *et al.* [15] apply MPTCP to improve the user experience on video streaming.

The core element of MPTCP design is the scheduler [28], which determines when and how to distribute packets to each sub-flow. MPTCP's default scheduler (minRTT) [26] sends packets through the available path with the smallest estimated Round-Trip Time (RTT). However, this scheduler does not take into account the path heterogeneity while there often exist different types and quality of paths in practical use and this is especially the case for wireless applications [17, 29]. The measurement of Alexa top-500 U.S. websites from Nikravesh *et al.* [24] shows that the difference in RTT between WiFi and Cellular paths is common and big. When RTTs in separate paths differ, the default scheduler will cause an out-of-order arrival of packets at the receiver side. Thus the memory requirements of MPTCP are much higher than those of conventional TCP. To alleviate this problem, opportunistic retransmission and penalization mechanism is proposed [26] and improved [25] along with the progress of the default scheduler.

Despite these efforts, we find that the complete gain from bandwidth aggregation of MPTCP is still far from being achieved. In our experiments conducted in the controlled lab environment (§ 2), we observe that when the host buffer is limited, the aggregation throughput is far smaller than two single-path TCP combined under

[*] Corresponding author

the same network and buffer settings. Sometimes it can only reach 40% the throughput of two single-path TCP together, which is even worse than a single-path TCP running over the best network interface.

Apart from the host buffer problem, we find that it also requires a big in-network buffer on the fast path to reach the full-bandwidth allowed by multiple paths. The in-network buffer requirement for the fast path is increased by 4X when using both paths compared with those for single-path TCP. Examining the problem carefully, we find that the default scheduler breaks down the ACK clocking [19] on the fast path, which gives rise to the burst sending of packets during fast sub-flow's transmission. Thus more in-network buffers are necessitated to hold the burst packets, otherwise the packets that cannot be stored have to be dropped, resulting in the throughput degradation on the fast path. This problem is more serious in the wireless networks where WiFi path is usually the fast one and has less buffer than that of the Cellular path [20]. Therefore when competing with single-path TCP, MPTCP is more likely to experience loss and the throughput will suffer. To the best of our knowledge, we are the first to identify the burst transmission pattern on the fast-path of MPTCP.

In this work, we propose a new scheduler to reduce both host buffer and in-network buffer requirements in MPTCP, called STMS (**S**lide **T**ogether **M**ultipath **S**cheduler). To solve the above two problems fundamentally, we need to reduce the out-of-order arrival. Our scheduler preallocates packets for the fast path and sends packets with larger sequence number through slow path so that packets can arrive at the receiver in order. This task appears to be straightforward, but it faces several challenges. As a matter of fact, there exists a ”visibility gap” between the sender and the receiver. Therefore, it is hard for the scheduler that runs at the sender to ensure that packets arrive in order at the receiver. A sender can choose to utilize the measurement of path condition to schedule the packets, but the network condition is fluctuated in nature. Consequently, it is hard to measure the delay and bandwidth accurately especially in wireless networks. Even under stable network conditions, we can only obtain RTT but not one-way delay (OWD) in a practical distributed network. To address these challenges, we design an adaptation scheme that exploits the intelligent transmission of feedback signal *Data ACK* existing in MPTCP to dynamically schedule packet transmissions.

We implemented STMS as a Linux kernel module (§ 4) and evaluated it extensively in both emulated and real Cellular/WiFi networks. The results show that, compared with state-of-the-art schedulers [23, 25], STMS achieves significantly higher performance under a wide range of buffer/network conditions. We highlight some of results as follows:

- Under stable network conditions, STMS can achieve higher throughput under any buffer conditions. When the host buffer is extremely limited, STMS can fall back to using the single-path TCP, while the default scheduler still uses slow path of MPTCP and its fast path suffers from significant throughput degradation. In this case, our scheduler can improve the throughput as much as 400%. When the host buffer is small, STMS can bring 15% improvement over the default scheduler. When in-network buffer is limited, STMS improves the throughput as much as 30% due to the reduction of the burst.

- Under varying network conditions, STMS also performs well, and brings 8% to 40% throughput improvement. Our adaptive scheduling scheme is reactive to network condition changes.

- In real world test, STMS can reduce the file downloading time by 20% even when the host buffer is big enough, proving that the limited in-network buffer does exist in real network and our scheme effectively alleviates the problem.

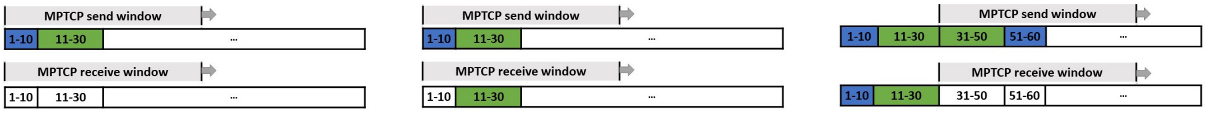
Overall, our results show that by strategically scheduling packet transmissions to reduce the out-of-order arrival, STMS can significantly improve the throughput of MPTCP under heterogeneous networks. The rest of the paper is organized as follows. Firstly, we analyze the reason that lead to the throughput degradation when using the default scheduler in § 2. Then in § 3, we present the design and analysis of STMS. Next we introduce our implementation of STMS in § 4. We further present our performance evaluations in a controlled-lab environment in § 5, and the results from the real-world test in § 6. Finally, related work is discussed in § 7 and we conclude the paper in § 8.

2 Background and Motivation

In this section, we first identify and analyze the problem associated with the in-network buffer. Then we demonstrate that host buffer problem still remains unsolved and discuss why the existing solution is still inadequate. We support our analyses with experiments conducted in the controlled lab environment.

2.1 Controlled experiment setup

In our experiment setup in fig. 2, we use two PCs running the version 0.92 kernel of MPTCP [7] as the client and server respectively. The client has two interfaces, and the server has one interface. Under this topology, MPTCP will establish two sub-flows. We use the decoupled TCP



(a) t = δ

(b) t = RTT_f

(c) t = RTT_s

Figure 1: Out-of-order arrival and burst transmissions on the fast path due to the use of default scheduler. Green (lighter) packets are sent/received through the fast path, while blue (darker) ones are sent/received through the slow path.

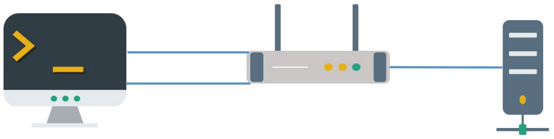


Figure 2: Topology setup

congestion control to obtain the best bandwidth aggregation effect [9]. Ethernet is used to simulate WiFi and LTE to avoid the interference of the wireless network. Client and server are connected through a router running OpenWrt. We use tc [30] in OpenWrt to regulate the bandwidth and latency of the two paths. The bandwidths of both paths are set to 30 Mbits and the loss rate is set to 0.01%. The in-network buffer is set to 50ms for Wifi path unless specified otherwise and 500ms for LTE path based on [20]. Both receiving and sending buffers are set to the Linux default size (6MB) unless specified otherwise. Only the RTT of slow path varies. In each network setup, we run iPerf 3 [18] 90s five times to measure the throughput.

2.2 Big in-network buffer requirement

Guido *et al.* [3] points out that it becomes more and more difficult to design the router with the buffer size equal to the bandwidth-delay product as the link speed increases. The router buffer is limited especially in the bottleneck of path. However, we find that the aggregated throughput is subject to the in-network buffer on the fast path as shown in fig. 3. For conventional TCP, 8ms network buffer (30KB) is enough to reach the full bandwidth. However, for MPTCP under different RTT paths, the in-network buffer requirement increases as much as 4X to fully unleash the power of the bandwidth aggregation. The bigger the difference of RTT between two paths, the bigger the in-network buffer of the fast path is needed.

When there is a space in the congestion window, the default MPTCP scheduler sends a set of packets with the smallest sequence numbers on the path with the smallest estimated RTT. This approach will produce the out-of-order arrivals at the receiver, as is elaborated in the following example (fig. 1). At the time $t = 0$, we assume the fast path is unavailable while the slow path has space,

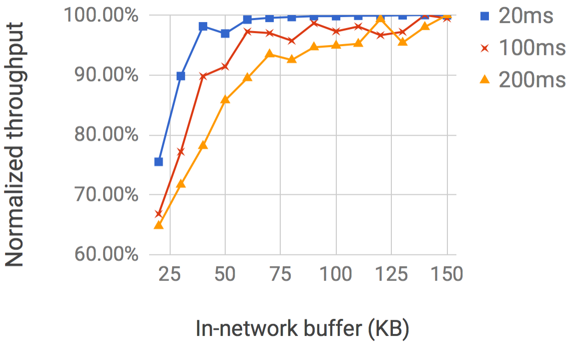
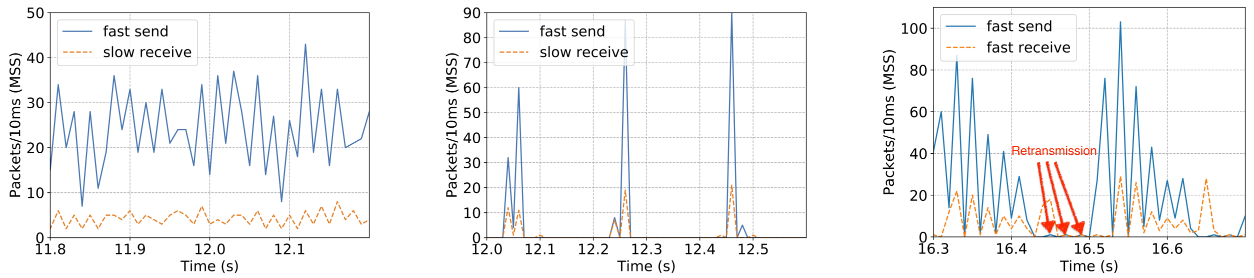


Figure 3: MPTCP throughput degradation when in-network buffer is limited

then packets 1-10 will be sent on the slow path. Later on at $t = \delta$, the fast path becomes available, packets 11-30 will be sent on the fast path. After the round-trip time of the fast path RTT_f, packets 11-30 arrive at the receiver but packets 1-10 do not, and the send/receive window is blocked as shown in fig. 1b.

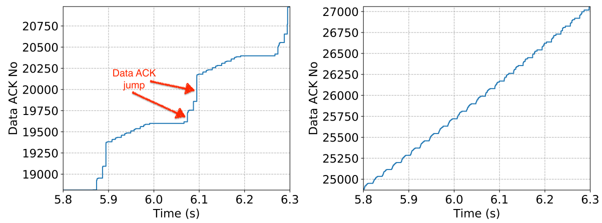
TCP has the delayed ACK mechanism [6] to avoid the overhead of sending ACK packets. It works as follows. Upon receiving a data packet, if it is in order, *i.e.,* the right edge of the receiving window advances, the receiver can choose to delay the sending of ACK hoping to piggy-back the ACK with other packets to send in the reverse direction. Nevertheless, RFC 1122 [6] suggests that each ACK acknowledge at most two packets regardless of whether the delayed ACK mechanism is used, *i.e.,* upon receiving two successive packets, an ACK must be sent. Thus, during the congestion avoidance phase, upon receiving one ACK, the sender's send window can have the space for at most two packets so that it can send at most two packets at a time. This is also known as ACK clocking.

In MPTCP, the semantics of the TCP send window is generalized. Instead of maintaining a separate window for each sub-flow, a single buffer pool is shared by all sub-flows at the MPTCP-level to avoid deadlock [26]. To remain compatible with TCP, MPTCP needs a separate ACK for MPTCP-level send window, called Data ACK. The delayed ACK mechanism of conventional TCP is adapted to Data ACK. When receiving data packets in-order on the MPTCP-level, Data ACK will still be



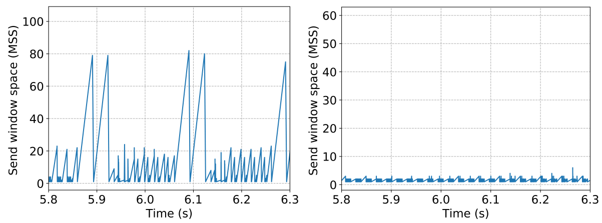
(a) $RTT_f = 20ms, RTT_f = 20ms$ w/o PR (b) $RTT_f = 20ms, RTT_s = 200ms$ w/o PR (c) $RTT_f = 20ms, RTT_s = 200ms$ w/ PR

Figure 4: Burst sending pattern of fast path



(a) $RTT_f = 20ms, RTT_s = 200ms$ (b) $RTT_f = 20ms, RTT_s = 20ms$

Figure 5: The Data ACK of fast path.



(a) $RTT_f = 20ms, RTT_s = 200ms$ (b) $RTT_f = 20ms, RTT_s = 20ms$

Figure 6: Send window free space jitter when RTT differs

generated every other packet. However, when receiving out-of-order packets on the MPTCP-level, it will not send a duplicate Data ACK immediately since out-of-order packets on the MPTCP-level is a normal behavior especially when paths are heterogeneous. The Data ACK won't be sent until the packets from the slow path reach the receiver, *i.e.,* the hole is filled. This Data ACK will acknowledge many packets sent from the fast path at the same time, thus fast path can send many packets at once, leading to the burst sending. In a nutshell, the out-of-order arrival of packets breaks down the ACK clocking effect of fast sub-flow, causing the burst sending behavior. This is shown in fig. 1c. At t = RTT_s, packets 1-10 arrive at the receiver and packets 1-30 are acknowledged to the sender. So both the send and receive window will progress with a large step, and packets 31-50 are sent from the fast path in a burst. Simply removing the delayed ACK mechanism can not solve the problem, since the MPTCP-level send window progress is still blocked by the late arriving packets from the slow path.

We conduct a simple experiment to demonstrate the burst transmission pattern of the fast sub-flow.

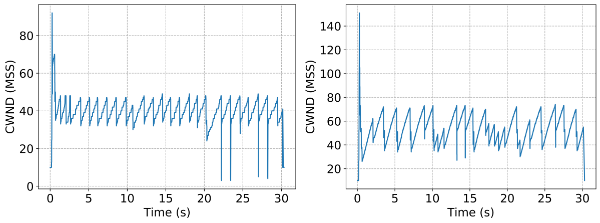
First we analyze the progress of Data ACK of the fast path from the trace file. The result is shown in fig. 5. Only when packets from the slow path arrive, the Data ACK can make a progress with a large step (fig. 5a). When RTT is the same, no sudden Data ACK progress happens. Then we check the free space in the send window of the fast path. We record the free space of send window when the sub-flow receive ACK (fig. 6). When RTT is the same, the ACK clocking is maintained like the conventional TCP (fig. 6b). However, when RTT over two paths significantly differs, the ACK clocking of the fast path is broken down. Consequently, MPTCP-level send window is stalled until the packets from the slow path arrive. As a result, the free space of the send window of the fast sub-flow will accumulate as we see in fig. 6a.

The sudden progress of MPTCP-level window and the big send window space of fast path will lead to the burst packet transmissions on the fast path as we see in fig. 4b. Compared to the sending behavior of the fast path when RTT is similar (fig. 4a), the fast sub-flow clearly demonstrates an on-off transmission pattern that leads to the burst of the fast sub-flow. When the network buffer is not big enough, the loss will happen and the Congestion window (CWND) is capped to a small value as we see in fig. 7a. When there is a difference between RTT on the two paths, the CWND of the fast path is capped around 40, compared to 60 when RTT values of two paths are the same. Note that there is usually a large space in fast path's CWND when RTT is different, so the area below the CWND line is actually bigger than the total throughput.

We also verify the loss rate under different in-network

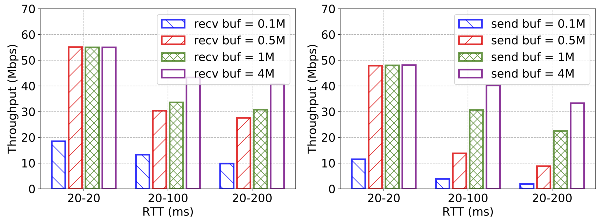
Table 1: Loss rate of fast path under different in-network buffer setting

in-network buffer/KB	observed loss rate	Fast path in MPTCP/Mbps	Single TCP/Mbps	Utilization
30	0.05%	12.1	28.4	42.76%
60	0.02%	20.8	28.4	73.50%
90	0.02%	25	28.4	88.34%
150	0.01%	26.5	28.4	93.64%



(a) $RTT_f = 20ms, RTT_s = 200ms$ (b) $RTT_f = 20ms, RTT_s = 20ms$

Figure 7: CWND of fast sub-flow is capped when in-network buffer is limited and RTTs are different.



(a) Receive buffer (b) Send buffer

Figure 8: MPTCP throughput degradation when host buffer is limited

buffer configurations (table 1). Note that this loss is more detrimental to the throughput than the random packet loss since each time CWND grows to certain value, the packet burst will exceed the in-network buffer size and the loss will happen. As a result, the throughput of the fast path is significantly limited. As shown in table 1, to reach the same throughput as that of single-path TCP, the buffer for two-path MPTCP has to be increased by 4X from 30KB to 150KB.

2.3 Big host buffer requirement

Another problem of MPTCP when RTTs are different is the big host buffer requirement. Let us denote the bandwidth of fast and slow sub-flow as B_f and B_s respectively. The one-way delay (OWD) values of both paths are denoted as OWD_f and OWD_s respectively. The RTTs of both paths are denoted as RTT_s and RTT_f. Raciu *et al.* [26] derives that the default scheduler buffer requirement is:

$$Buf(default) = (B_f + B_s) \times RTT_s \tag{1}$$

From eq. (1) we can see that the key to reducing the buffer requirement is to reduce the effective RTT of the connection *i.e.,* to acknowledge packet as soon as possible so that the buffer can get freed.

Raiciu *et al.* [26] proposed the penalize and opportunistic retransmission mechanism (PR) to deal with the host buffer problem. When it detects the left edge of the send window blocks the sending of packets, it will retransmit the packet from the fast path. To avoid constant retransmissions, it will penalize the slow sub-flow by halving the CWND of slow sub-flow. This approach does improve the throughput when the receiving buffer is limited because the blocked packets will get retransmitted and acknowledged through the fast path. Thus the effective RTT of MPTCP connection can be reduced to $OWD_s + OWD_f$ ideally. As shown in fig. 4c, the packets marked get retransmitted and the new Data ACK goes back through the fast path so many packets can be sent. Due to its opportunistic nature, the PR scheme can not always reduce the RTT to the optimal value. Moreover, the retransmission wastes the bandwidth. fig. 4c also shows that the retransmission does not change the fast path's burst transmission pattern, as the Data ACK coming back through the fast path will still acknowledge many packets. Hence the in-network buffer requirement issue remains unsolved which is also shown in the in-network buffer requirement measurement. Enabling retransmission doesn't improve the throughput at all.

Despite the PR approach, the throughput of MPTCP is still unsatisfactory as shown in fig. 8. When evaluating the receiving buffer, the sending buffer size is set to the default value of Linux which is 6 MB (Big enough under our network setting). As we can see in fig. 8a: the bigger the RTT difference of two paths, the more receiving buffer is needed to get full bandwidth aggregation effect of MPTCP. The same conclusion goes to the sending buffer as is shown in fig. 8b.

Actually, the burst sending pattern can be fixed by adding traditional pacing to the fast path. Linux has supported pacing in tc qdisc fq [13] since the version 2.4. However the pacing rate argument needs to be manually tuned according to the bandwidth of the network path. New congestion control algorithm Bottleneck Bandwidth and Round-trip propagation time (BBR) [8] incorporates the pacing and can set the pacing rate equal to the measured bottleneck bandwidth automatically. However the congestion control of MPTCP needs to be fair with the conventional TCP. Many active research efforts [5, 21, 31] have been put into developing the coupled congestion control algorithms of MPTCP, but there is no coupled BBR congestion control for MPTCP yet. To put it another way, this approach is not congestion control agnostic. Besides, the pacing can not solve the big host buffer requirement issue. Pacing works only on sub-RTT level and thus the packet sent from the slow path will still arrive later than packets sent from the fast path. We verify that by adding pacing manually and it turns out that the throughput is not improved at all.

3 Design

The root cause of the throughput degradation is the stall and sudden progress of MPTCP-level send window. To solve both host buffer and in-network buffer problems, we need to restore the ACK clocking for MPTCP. To achieve this, packets need to arrive in order at the receiver.

In this section, we first present our algorithm design, then derive the size required from the host-buffer for no-blocked transmissions, and finally compare the transmission latency of different schemes.

3.1 STMS algorithm

We propose STMS to schedule the packets strategically so that they arrive in order. This solution is congestion control agnostic which allows for separate evolving of congestion control algorithm and scheduler algorithm.

Scheduler algorithm The core idea of our scheduler is to buffer packets for the fast sub-flow and assign packets with larger sequence numbers to the slow sub-flow so that they arrive in order. The running process of our algorithm is shown on fig. 9 and algorithm 1. The scheduler runs when at least one of paths is available to send packets. The fast sub-flow always sends the packets with the smallest set of sequence numbers in the buffer. As illustrated in fig. 9a, the slow sub-flow sends packets with bigger sequence numbers. Rather than taking packets whose numbers are right after those transmitted on the fast path, it leaves a sequence gap for the fast path to send the corresponding packets in the future. By the time the packets from the slow path arrive, all packets from the fast path which have smaller sequence numbers should have already arrived (fig. 9b). Since packets arrive in order, the normal ACK clocking is ensured, so there are no burst transmissions on the fast path and the send/receive window will not be blocked.

Algorithm 1 Slide Together Scheduler

```
procedure ST_SCHEDULE($unsentPackets$)    ▷ Scheduler runs when one of sub-flow is available
    if Fast sub-flow has space in send window then
        $Fast\ sub-flow \leftarrow$ unsentPackets[0]
    elseSlow sub-flow has space in send window
        $Slow\ sub-flow \leftarrow$ unsentPackets[$Gap$]
    end if
end procedure
```

The key parameter of the scheduler algorithm is *Gap*, which is the number of packets pre-allocated for fast path to send. The efficiency of the scheduler algorithm depends on the accuracy of the gap value. Any deviation from the true value will cause out-of-order arrival of packets.

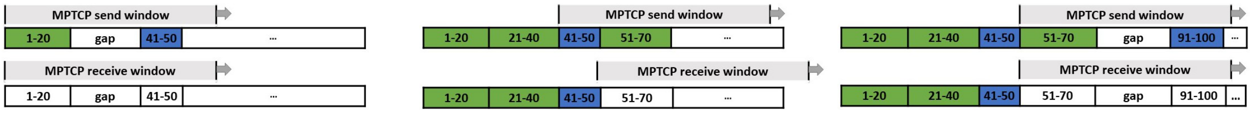
The naive way to get the gap value is to utilize the measurement of path conditions. If we can measure network conditions accurately, then we can derive the true gap value in the following way. It takes $OWD_f + \frac{Gap}{B_f}$ for all packets in the gap to arrive at the receiver through the fast path, where B_f is the bandwidth of the fast path. This should be equal to OWD_s so that the first packet from the slow path arrives at the same time as packets from the fast path. Then we have the true Gap value:

$$True\_Gap = B_f \times (OWD_s - OWD_f) \qquad (2)$$

However, the naive solution has two fundamental flaws. One is that the one-way delay of path can not be measured accurately. We can not assume the uplink delay and the downlink delay are the same on both paths. If we modify the protocol to carry the one-way delay information, it may cause other compatibility problem with middleboxes [16]. The other one is that the bandwidth of the path can not be measured accurately, especially when the in-network buffer is limited. Because of the burst sending pattern of fast sub-flow, it can never reach the actual available bandwidth. So we design the feedback-based gap adjustment scheme to adjust the value of gap more accurately and quickly.

Key insight: Every out-of-order packet in the receiver will generate duplicate Data ACK or burst Data ACK.

What STMS actually does is moving stalled packets from the receiver to the sender (*i.e.,* the packets inside gap). The out-of-order packets will be acknowledged at one time when packets from the slow path arrive to fill in the hole. The number of packets acknowledged by Data ACK reflects the degree of out-of-order arrival of packets. Accordingly we can use this as the feedback signal to adjust the gap value. Since Data ACK is presented in MPTCP, our scheme remains compatible with current MPTCP. In addition, this scheme does not require



(a) t = 0

(b) t = $2RTT_f$

(c) t = RTT_s

Figure 9: Demonstration of STMS, with green and blue for packets over fast and slow paths respectively. Let $RTT_s = 3RTT_f$ and assume the uplink delay and downlink delay are symmetric, then we have $gap = CWND_f$ according to eq. (2). Note that slow path always send packets with sequence numbers bigger than those of the fast path.

any modification at the receiver, which further eases the deployment process.

How to adjust The gap adjustment algorithm is shown in algorithm 2. Let $delta\_gap$ and $adjust\_interval$ be the gap adjustment step and interval. When the gap is smaller than the true gap value, the packets from the slow path arrive late and the send window of MPTCP-level will be stalled by packets sent from the slow path. Therefore the left edge of the send window is determined by unacknowledged packets from the slow path. Symmetrically, when the gap is bigger than the true gap value, the left edge of the send window will be determined by the packets sent from the fast path. Each time we receive a Data ACK, we first calculate how many packets this Data ACK acknowledges ($data\_acked$). If the $data\_acked$ is bigger than 2, we will adapt the gap value. We check the packet of the left edge of the send window. If the packet is the first one sent from the slow path, $delta\_gap = data\_acked$; otherwise, $delta\_gap = -data\_acked$. We use the Exponentially Weighted Moving Average (EWMA) of $delta\_gap$ over $adjust\_interval$ to adjust the gap value. The $adjust\_interval$ is a tunable parameter, which determines how fast the gap adjustment can react to the network change. Setting it too small will cause the gap overshoot and oscillation since the previous gap adjustment has not taken into effect yet. However, setting it too big leads to the slow convergence time.

Algorithm 2 Gap Adjustment Algorithm

```
procedure GAP_ADJUST($data\_acked$)    ▷ This function gets called when receiving Data ACK
    if $data\_acked > 2$ then
        $send\_una \leftarrow$ left edge of MPTCP-level send window
        if $send\_una$ was sent from slow path then
            $delta\_gap = data\_acked$
        else$send\_una$ was sent from fast path
            $delta\_gap = -data\_acked$
        end if
    end if
    $gap+ = EWMA(delta\_gap, adjust\_interval)$
end procedure
```

3.2 Analysis of host buffer size requirement

At a first glance, buffering packets for the fast path may require a big buffer on the sender side. However, we can prove that when both paths are fully utilized, the send buffer requirement is actually less than that of the default scheduler without PR (eq. (1)).

When using STMS the send buffer consists of three parts:

1. sent but unacknowledged packets from the fast path: $B_f \times RTT_f$
2. sent but unacknowledged packets from the slow path: $B_s \times (OWD_s + OWD_f)$ (Data packet is sent through the slow path, but ACK returns from the fast path).
3. buffered packets for the fast path *i.e.,* $True\_Gap$ (eq. (2))

By adding these three parts together, we get the buffer requirement of STMS:

$$Buf(STMS) = (B_f + B_s) \times (OWD_S + OWD_f) \qquad (3)$$

This is smaller than $Buf(default)$ (eq. (1)). It also reveals that STMS reduces the effective RTT of the MPTCP connection to $OWD_s + OWD_f$, which is the smallest RTT when both paths are utilized. Thus our STMS can reduce the send buffer requirement.

If we take into consideration the opportunistic retransmission, then in the ideal case, upon receiving the late arrival packet from the slow path, the Data ACK of the retransmitted packet will go back through the fast path. Therefore the effective RTT of the MP connection is reduced to $OWD_s + OWD_f$, which is the same as STMS and the buffer requirement is also the same.

When the host buffer is between [$Buf(STMS)$, $Buf(default)$], both retransmission and STMS can improve the throughput. But STMS can always achieve the optimal throughput by ensuring RTT of MP connection to be the minimum.

If the host buffer is smaller than $Buf(STMS)$, neither STMS nor default scheduler can get the full bandwidth aggregation. In this case, STMS will prefer to use the fast path. The slow path will be used only if it will not cause the blocking. The buffer requirement to take advantage of the use of the slow path is:

$$\begin{aligned} Buf(fallback) &= RTT_f \times B_f + Gap \\ &= B_f \times (OWD_s + OWD_f) \end{aligned} \tag{4}$$

When the host buffer is between [$Buf(fallback)$, $Buf(STMS)$], STMS will use the fast path first, as the slow path will not block the fast path. However, for the default scheduler, the slow path will get the frequent use, which would trigger the retransmission of packets. This will further lead to the goodput degradation and the big end-to-end latency.

What if the host buffer is even smaller than $Buf(fallback)$? Then STMS will fall back to the single TCP over the fast path. But the default scheduler will still send some packets from the slow path, which pushes the effective RTT of MPTCP connection to at least $OWD_s + OWD_f$. Thus the throughput will be even worse than the bandwidth B_f allowed by the fast path alone. Actually, in this case, falling back to the single-path TCP is the optimal choice.

So our scheduler can get the optimal throughput across all range of host buffer sizes.

3.3 Analysis of latency

It seems that STMS will cause the inflation of transmission latency because it holds packets in the gap longer than the default scheduler. However, it also reduces the time duration for the packets to be stalled in the receiver. Using both types of scheduler, the end-to-end latency of packets sent from the slow path is OWD_s. The latency of the fast path is $OWD_f + Delay(Stalled)$. For each packet sent from the fast path, it is either stalled at the receiver buffer or held at the send buffer. $Delay(Stalled)$ remains the same. Therefore STMS does not increase the average end-to-end latency of packets.

4 Implementation

We implement STMS in the Linux kernel based on MPTCP version 0.92 from [7]. The algorithm 1 is implemented as a scheduler module.

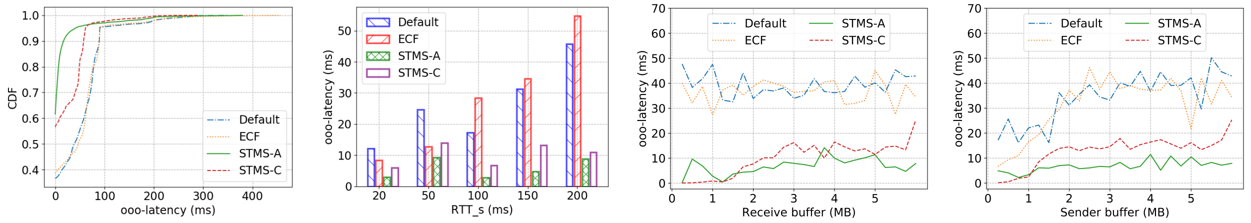
The MPTCP scheduler will run when two types of event happen. The first type of event happens when ACK returns from one of the sub-flows, which means there will be space in the sub-flow send window. The second type of event happens when application sends more data. The scheduler makes the decision every data segment. For each segment pushed in by the application, the scheduler will determine which sub-flow to send the packet. This framework of scheduler limits how we can implement our scheduler, since we can not access an arbitrary segment inside the send buffer. To remain compatible with this framework. We implement our scheduler as follows.

When the scheduler picks the next segment to send, we first check if the fast path is available, *i.e.,* there is space in the send window. If the space is available, then send the packet over the fast path; otherwise, we check if the slow path is available. If it is available, we find the packet according to the gap value, *i.e.,* jump across the gap packets to find the packets to send over the slow path. If the packet does not exist, that means either the packet is out of the right boundary of the send window or the application has not pushed enough data yet, in either of which case we skip this round of scheduling. Note that we need to mark the packets sent from the slow path so that we can skip the out-of-order packets when finding the next packet to send from the slow path. To avoid traversing the send buffer from the beginning each time, we save the pointer of the last send packet of the slow path as the beginning point for the next search.

We implement two variants of STMS: STMS-C ("Calculation") and STMS-A ("Adjustment"). They both preallocate packets for the fast sub-flow so that packets can arrive in-order at the receiver side (§ 3.1). They differ in how they obtain the gap value. Each time a packet is sent from the slow path, STMS-C extracts the bandwidth estimation of smoothed RTT from subflow TCP's algorithm and calculates the gap value (assume the uplink and downlink delay are symmetric). For STMS-A the Data ACK process function is modified to calculate $delta\_gap$ according to algorithm 2.

5 Evaluation in a controlled lab environment

In this section, we test both STMS-C and STMS-A in a controlled lab setting, which allows us to evaluate the performance across a wide variety of network conditions. We compare our scheduler with the default scheduler with PR (denoted as Default thereafter) and ECF [23]. ECF uses the send buffer length to estimate the flow complete time(FCT) of using each path. If using slow path will cause inflation of FCT, it will wait for fast sub-flow. However, for elephant flow, the send buffer is full for most of the time. Only when flow is about to finish, the send buffer length can be small enough to wait for fast sub-flow. Besides, when calculating the FCT, ECF does not take into account the one way delay thus it is not able to achieve accurate in-order arrival. STMS schedules the packets out-of-orderly to achieve the in-order arrival regardless of the send buffer status so STMS can outperform ECF. For apples-to-apples comparison, we port the ECF



(a) OOO delay distribution

(b) Varying slow path latency

(c) Varying receive buffer

(d) Varying send buffer

Figure 10: Out-of-order latency of different schedulers

code [10] to the same MPTCP version as our scheduler. The experiment setup is the same as § 2.1. We tune the $adjust\_interval$ of STMS-A to $\frac{RTT_s+RTT_f}{2}$ according to the analysis in § 3.1. The parameter of ECF is chosen as the same as [23].

5.1 Microbenchmarks

We first focus on some micro-benchmarks to see whether our scheduler can accomplish the design goal.

5.1.1 Reducing the out-of-order arrival at the receiver side

We first investigate whether our scheduler can achieve the in-order arrival at the receiver side. We use the out-of-order (OOO) delay as the metric. The OOO delay of a packet is defined as the time difference between when a packet arrives at the receiver to when the packet can be submitted to the application (*i.e.,* all packets with the smaller data sequence numbers have already arrived).

fig. 10a shows the CDF of the OOO delay of each packet with different schedulers. STMS-C and STMS-A can both achieve smaller OOO delay than Default and ECF. The largest OOO delay is around 300ms. Default effectively pushes the OOO delay of most packets sent from fast path to OWD_s. ECF sends tail packets out-of-orderly so it can reduce OOO delay for those tail packets. However since we transmit many packets for a test, this delay reduction is negligible. STMS-A can effectively push the OOO delay close to zero.

We vary the latency of slow path and calculate average of OOO latency. The result of one experiment is shown in fig. 10b. When paths become more heterogeneous, both Default and ECF have larger OOO delay. However STMS-A can keep its average OOO delay at a very small value. When RTT of two paths are similar(20ms and 50ms), ECF, STMS-C and STMS-A get close performance.

We also test the OOO delay under different host buffer sizes. The result is shown in fig. 10. It demonstrates that both STMS-A and STMS-C can effectively reduce the OOO delay regardless of the host buffer size, and the gain is larger when the host buffer sizes are larger with more packets stalled at the receiver. For ECF, only when the send buffer is very small, it wait for fast sub-flow to reduce the OOO delay.

5.1.2 Reducing the burst on the fast path

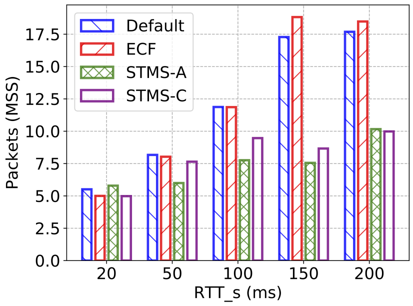
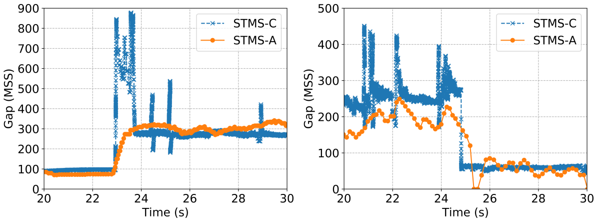


Figure 11: Burstness of all four schedulers under different path latencies

We now study whether our scheduler can reduce the burst of the fast path thus the in-network buffer requirement accordingly. We print the CWND free space of the fast sub-flow when it receives ACK. Since all schedulers try to fill this free space, the peak value of CWND free space reflects the burst of fast path. The average CWND free space throughout the running time is used as a metric of burstness of fast subfow. The result is shown in fig. 11. The trend is the same as fig. 10b. Our scheduler can reduce the burstness of the fast path and makes it close to that of the single-path TCP.

5.1.3 Gap adjustment is reactive to network change

To understand how STMS-A handles dynamic network conditions, we change the network conditions in the middle of MPTCP flow and record the gap value changes around the condition changing point. Recall that the true gap value is calculated using eq. (2). It is affected by the accuracy of the measurement of the fast path bandwidth and the one-way delay of both fast path and slow path. We choose to change the latency of the fast path and slow path to demonstrate how our Gap adjustment algorithm reacts to the network change. In fig. 12a, the latency of the fast path changes from 20ms to 5ms. Therefore there will be many ACK packets and the fast path can send a lot of packets out which leads to the wrong estimation of the bandwidth. Thus, we see the peak of the gap calculated value. However, our gap adjustment algorithm can converge to the new value smoothly. In fig. 12b, the latency of the slow path changes from 200ms to 100ms. Again STMS-A converges to the new value smoothly and fast.



(a) Gap increase

(b) Gap decrease

Figure 12: Gap adjustment is reactive to network change

5.2 Macrobenchmark: improving aggregated throughput

We then investigate how our scheduler improves the aggregated throughput under different buffer sizes setting.

Stable network condition We begin by investigating whether our scheduler improves the throughput when the network condition is stable. fig. 13 shows the result. When the in-network buffer is limited, our scheduler can improve the throughput by about 30% compared to Default and ECF. When the host buffer is extremely limited, our scheduler falls back to single path TCP and outperforms Default as analyzed in § 3.2. When the host buffer is big enough(>4MB), there is no blocking, so all four schedulers can get the full bandwidth aggregation effect. The STMS-C and STMS-A produce analogous results under the same buffer settings, which indicates the gap adjustment algorithm can track the true gap value precisely.

We then vary the latency of slow path. Both STMS-C and STMS-A improve the aggregation throughput over Default and ECF. We pick the improvement of STMS-A over Default as an example. fig. 14 shows the throughput of STMS-A normalized relative to that of Default. The improvement become more prominent as the paths become more heterogeneous and the buffer gets smaller.

Varying bandwidth Then we investigate the performance of our scheduler under network fluctuations. Here we change the bandwidth of both path randomly every 10 seconds. The bandwidth value is chosen from set {5, 10, 20, 30} Mbps. We generate 5 network configurations using different random seeds and run test five times for each network configurations.

fig. 15a shows the average throughput of four schedulers for each configuration. Note the error bar indicates the variability of the same configuration. Our scheduler outperforms other schedulers in every configuration. STMS-A performs even better than the STMS-C. This indicates that STMS-A is more adaptive to network fluctuations than the STMS-C.

Varying latency We simulate the varying latency condition using tc netem module. Both paths' latency is changed every 10 seconds, and the stddev of latency is set to 40% of the mean latency. We generate five latency configurations and run the test five times (fig. 15b). Similar to the bandwidth change scenario, STMS-A always performs best.

6 Evaluation in the wild

We next evaluate our scheduler in more realistic environments. The server is deployed in Aliyun [1] and has only one interface. The client is located inside our campus and connects to the server through WiFi and LTE. The China Telecom LTE cellular network incurs higher delay than the WiFi network. The average bandwidth and latency characteristic of each path are shown in table 2. We use tc to add extra latency on LTE path.

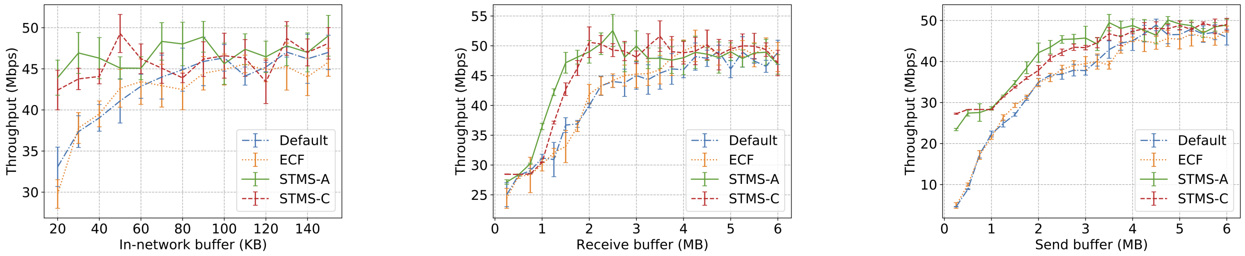
Table 2: Path characteristics

	Bandwidth(Mbps)	Latency(ms)
WiFi	40	50
LTE	30	70

We compare our scheduler against Default and ECF. We measure the download time of 200MB file of different schedulers. For each latency setting, we run the test five times. The result is shown in fig. 16. Both STMS-C and STMS-A outperform Default and ECF. The STMS-A can get the best performance, reducing the file download time by as much as 20% over Default.

7 Related work

There are many studies on the improvement of MPTCP scheduler. To solve the host buffer problem, Raiciu *et al.* [26] propose the PR mechanism. Ferlin *et al.* [11] propose the Blocking Estimation-based Scheduler (BLEST) which aims to prevent the blocking by reducing the usage of slow path even if it has available CWND space. Both schedulers try to restrict the use of the slow path to alleviate the need of big host buffer resulting in the under-utilization of slow path.

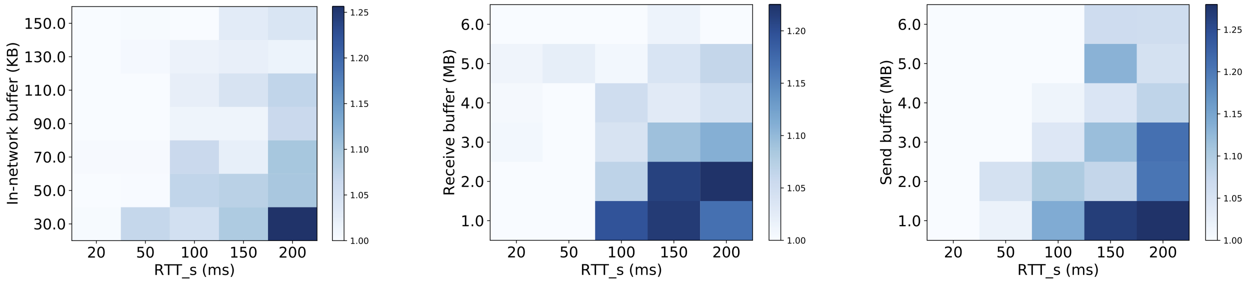


(a) Varying in-network buffer

(b) Varying receive buffer

(c) Varying send buffer

Figure 13: Throughput of different schedulers

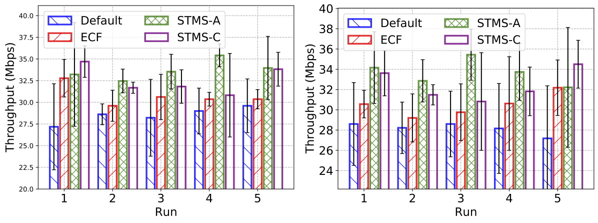


(a) Varying in-network buffer

(b) Varying receive buffer

(c) Varying send buffer

Figure 14: STMS-A throughput normalized by Default



(a) Bandwidth change

(b) Latency change

Figure 15: Throughput under dynamic network conditions

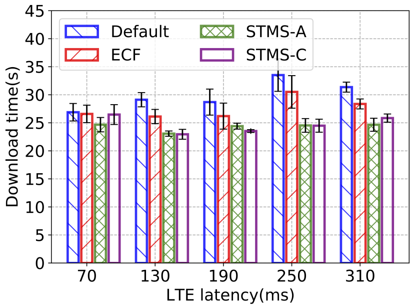


Figure 16: Average file download time in the wild (lower is better)

Kuhn *et al.* [22] propose DAPS to address the RTT difference of two paths. But it only considers the stable CWND and the scheduler running interval is very big thus can not react to the dynamic network changes.

Lim *et al.* [23] propose ECF which outperforms both DAPS and BLEST. But it only considers the tail packets.

Guo *et al.* [14] propose a new scheduler to balance two sub-flow completion time by sending packets inside a ”chunk” in the opposite direction. Nonetheless, this approach will require a huge host buffer to store the whole chunk.

8 Conclusion

In this work, we identify a new root cause of MPTCP throughput degradation under heterogeneous path conditions. We propose STMS to effectively alleviate the problems due to the limitation in the host buffer size and in-network buffer size. Our experimental results show that STMS outperforms state-of-the-art schedulers in diverse network and buffer settings, especially when the path heterogeneity is large.

Acknowledgments. We thank the anonymous reviewers and our shepherd Theophilus Benson for their feedback on the paper. This work is supported by National Key R&D Program of China under Grant 2017YFB1010002, National 863 project (no. 2015AA015701) and a joint research with Protocol Research Lab, Huawei Technologies.

References

[1] Alibaba Cloud. https://www.alibabacloud.com/.

[2] AGACHE, A., DEACONESCU, R., AND RAICIU, C. Increasing datacenter network utilisation with grin. In *NSDI* (2015), pp. 29–42.

[3] APPENZELLER, G., KESLASSY, I., AND MCKEOWN, N. *Sizing router buffers*, vol. 34. ACM, 2004.

[4] APPLE Opens Multipath TCP In iOS11. http://www.tessares.net/highlights-from-advances-in-networking-part-1/.

[5] Balanced Linked Adaptation Congestion Control Algorithm for MPTCP. https://tools.ietf.org/html/draft-walid-mptcp-congestion-control-00.

[6] BRADEN, R. Requirements for internet hosts-communication layers.

[7] C. PAASCH, S. BARRE, E. A. Multipath TCP in the Linux Kernel. http://www.multipath-tcp.org.

[8] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. Bbr: Congestion-based congestion control. *Queue 14*, 5 (2016), 50.

[9] DENG, S., NETRAVALI, R., SIVARAMAN, A., AND BALAKRISHNAN, H. Wifi, lte, or both?: Measuring multi-homed wireless internet performance. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), ACM, pp. 181–194.

[10] ECF implementation in old MPTCP version. http://www.cs.umass.edu/~ylim/mptcp_ecf.

[11] FERLIN, S., ALAY, Ö., MEHANI, O., AND BORELI, R. Blest: Blocking estimation-based mptcp scheduler for heterogeneous networks. In *IFIP Networking Conference (IFIP Networking) and Workshops, 2016* (2016), IEEE, pp. 431–439.

[12] FORD, A., RAICIU, C., HANDLEY, M., AND BONAVENTURE, O. Tcp extensions for multipath operation with multiple addresses. Tech. rep., 2013.

[13] manpage of linux tc-fq . https://www.systutorials.com/docs/linux/man/8-tc-fq/.

[14] GUO, Y. E., NIKRAVESH, A., MAO, Z. M., QIAN, F., AND SEN, S. Accelerating multipath transport through balanced subflow completion. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (2017), ACM, pp. 141–153.

[15] HAN, B., QIAN, F., JI, L., GOPALAKRISHNAN, V., AND BEDMINSTER, N. Mp-dash: Adaptive video streaming over preference-aware multipath. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies* (2016), ACM, pp. 129–143.

[16] HONDA, M., NISHIDA, Y., RAICIU, C., GREENHALGH, A., HANDLEY, M., AND TOKUDA, H. Is it still possible to extend tcp? In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (2011), ACM, pp. 181–194.

[17] HUANG, J., QIAN, F., GERBER, A., MAO, Z. M., SEN, S., AND SPATSCHECK, O. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), ACM, pp. 225–238.

[18] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/iperf-download.php.

[19] JACOBSON, V. Congestion avoidance and control. In *ACM SIGCOMM computer communication review* (1988), vol. 18, ACM, pp. 314–329.

[20] JIANG, H., WANG, Y., LEE, K., AND RHEE, I. Tackling bufferbloat in 3g/4g networks. In *Proceedings of the 2012 ACM conference on Internet measurement conference* (2012), ACM, pp. 329–342.

[21] KHALILI, R., GAST, N., POPOVIC, M., UPADHYAY, U., AND LE BOUDEC, J.-Y. Mptcp is not pareto-optimal: performance issues and a possible solution. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012), ACM, pp. 1–12.

[22] KUHN, N., LOCHIN, E., MIFDAOUI, A., SARWAR, G., MEHANI, O., AND BORELI, R. Daps: Intelligent delay-aware packet scheduling for multipath transport. In *Communications (ICC), 2014 IEEE International Conference on* (2014), IEEE, pp. 1222–1227.

[23] LIM, Y.-S., NAHUM, E. M., TOWSLEY, D., AND GIBBENS, R. J. Ecf: An mptcp path scheduler to manage heterogeneous paths. In *Proceedings of the 2017 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems* (2017), ACM, pp. 33–34.

[24] NIKRAVESH, A., GUO, Y., QIAN, F., MAO, Z. M., AND SEN, S. An in-depth understanding of multipath tcp on mobile devices: measurement and system design. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking* (2016), ACM, pp. 189–201.

[25] PAASCH, C., KHALILI, R., AND BONAVENTURE, O. On the benefits of applying experimental design to improve multipath tcp. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies* (2013), ACM, pp. 393–398.

[26] RAICIU, C., PAASCH, C., BARRE, S., FORD, A., HONDA, M., DUCHENE, F., BONAVENTURE, O., AND HANDLEY, M. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 29–29.

[27] In Korean, Multipath TCP is pronounced GIGA Path. http://blog.multipath-tcp.org/blog/html/2015/07/24/korea.html.

[28] Why is the Multipath TCP scheduler so important ? http://blog.multipath-tcp.org/blog/html/2014/03/30/why_is_the_multipath_tcp_scheduler_so_important.html.

[29] SOMMERS, J., AND BARFORD, P. Cell vs. wifi: on the performance of metro area mobile connections. In *Proceedings of the 2012 ACM conference on Internet measurement conference* (2012), ACM, pp. 301–314.

[30] tc: Linux Advanced Routing and Traffic Control. http://lartc.org/lartc.html.

[31] WISCHIK, D., RAICIU, C., GREENHALGH, A., AND HANDLEY, M. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI* (2011), vol. 11, pp. 8–8.

Pantheon: the training ground for Internet congestion-control research

Francis Y. Yan[†], Jestin Ma[†], Greg D. Hill[†], Deepti Raghavan[¶], Riad S. Wahby[†],
Philip Levis[†], Keith Winstein[†]

[†]Stanford University, [¶]Massachusetts Institute of Technology

Abstract

Internet transport algorithms are foundational to the performance of network applications. But a number of practical challenges make it difficult to evaluate new ideas and algorithms in a reproducible manner. We present the Pantheon, a system that addresses this by serving as a community "training ground" for research on Internet transport protocols and congestion control (https://pantheon.stanford.edu). It allows network researchers to benefit from and contribute to a common set of benchmark algorithms, a shared evaluation platform, and a public archive of results.

We present three results showing the Pantheon's value as a research tool. First, we describe a measurement study from more than a year of data, indicating that congestion-control schemes vary dramatically in their relative performance as a function of path dynamics. Second, the Pantheon generates calibrated network emulators that capture the diverse performance of real Internet paths. These enable reproducible and rapid experiments that closely approximate real-world results. Finally, we describe the Pantheon's contribution to developing new congestion-control schemes, two of which were published at USENIX NSDI 2018, as well as data-driven neural-network-based congestion-control schemes that can be trained to achieve good performance over the real Internet.

1 Introduction

Despite thirty years of research, Internet congestion control and the development of transport-layer protocols remain cornerstone problems in computer networking. Congestion control was originally motivated by the desire to avoid catastrophic network collapses [22], but today it is responsible for much more: allocating capacity among contending applications, minimizing delay and variability, and optimizing high-level metrics such as video rebuffering, Web page load time, the completion of batch jobs in a datacenter, or users' decisions to engage with a website.

In the past, the prevailing transport protocols and congestion-control schemes were developed by researchers [18, 22] and tested in academic networks or other small testbeds before broader deployment across the Internet. Today, however, the Internet is more diverse, and studies on academic networks are less likely to generalize to, e.g., CDN nodes streaming video at 80 Gbps [26], smartphones on overloaded mobile networks [8], or security cameras connected to home Wi-Fi networks.

As a result, operators of large-scale systems have begun to develop new transport algorithms in-house. Operators can deploy experimental algorithms on a small subset of their live traffic (still serving millions of users), incrementally improving performance and broadening deployment as it surpasses existing protocols on their live traffic [1, 7, 24]. These results, however, are rarely reproducible outside the operators of large services.

Outside of such operators, research is usually conducted on a much smaller scale, still may not be reproducible, and faces its own challenges. Researchers often create a new testbed each time—interesting or representative network paths to be experimented over—and must fight "bit rot" to acquire, compile, and execute prior algorithms in the literature so they can be fairly compared against. Even so, results may not generalize to the wider Internet. Examples of this pattern in the academic literature include Sprout [42], Verus [43], and PCC [12].

This paper describes the **Pantheon**: a distributed, collaborative system for researching and evaluating end-to-end networked systems, especially congestion-control schemes, transport protocols, and network emulators. The Pantheon has four parts:

1. a software library containing a growing collection of transport protocols and congestion-control algorithms, each verified to compile and run by a continuous-integration system, and each exposing the same interface to start or stop a full-throttle flow,

2. a diverse testbed of network nodes on wireless and wired networks around the world, including cellular networks in Stanford (U.S.), Guadalajara (Mexico), São Paulo (Brazil), Bogotá (Colombia), New Delhi (India), and Beijing (China), and wired networks in all of the above locations as well as London (U.K.), Iowa (U.S.), Tokyo (Japan), and Sydney (Australia),
3. a collection of network emulators, each calibrated to match the performance of a real network path between two nodes, or to capture some form of pathological network behavior, and
4. a continuous-testing system that regularly evaluates the Pantheon protocols over the real Internet between pairs of testbed nodes, across partly-wireless and all-wired network paths, and over each of the network emulators, in single- and multi-flow scenarios, and publicly archives the resulting packet traces and analyses at https://pantheon.stanford.edu.

The Pantheon's calibrated network emulators address a tension that protocol designers face between experimental realism and reproducibility. Simulators and emulators are reproducible and allow rapid experimentation, but may fail to capture important dynamics of real networks [15, 16, 31]. To resolve this tension, the Pantheon generates network emulators calibrated to match real Internet paths, graded by a novel figure of merit: their accuracy in matching the performance of a set of transport algorithms. Rather than focus on the presence or absence of modeled phenomena (jitter, packet loss, reordering), this metric describes how well the end-to-end performance (e.g., throughput, delay, and loss rate) of a set of algorithms, run over the emulated network, matches the corresponding performance statistics of the same algorithms run over a real network path.

Motivated by the success of ImageNet [11, 17] in the computer-vision community, we believe a common reference set of runnable benchmarks, continuous experimentation and improvement, and a public archive of results will enable faster innovation and more effective, reproducible research. Early adoption by independent research groups provides encouraging evidence that this is succeeding.

Summary of results:

- Examining more than a year of measurements from the Pantheon, we find that transport performance is highly variable across the type of network path, bottleneck network, and time. There is no single existing protocol that performs well in all settings. Furthermore, many protocols perform differently from how their creators intended and documented (§4).
- We find that a small number of network-emulator parameters (bottleneck link rate, isochronous or memoryless packet inter-arrival timing, bottleneck buffer size, stochastic per-packet loss rate, and propagation delay) is sufficient to replicate the performance of a diverse library of transport protocols (with each protocol matching its real-world throughput and delay to within 17% on average), in the presence of both natural and synthetic cross traffic. These results go against some strains of thought in computer networking, which have focused on building detailed network emulators (with mechanisms to model jitter, reordering, the arrival and departure of cross traffic, MAC dynamics, etc.), while leaving the questions open of how to configure an emulator to accurately model real networks and how to quantify the emulator's overall fidelity to a target (§5).
- We discuss three new approaches to congestion control that are using the Pantheon as a shared evaluation testbed, giving us encouragement that it will prove useful as a community resource. Two are from research groups distinct from the present authors, and were published at USENIX NSDI 2018: Copa [2] and Vivace [13]. We also describe our own data-driven designs for congestion control, based on neural networks that can be trained on a collection of the Pantheon's emulators and in turn achieve good performance over real Internet paths (§6).

2 Related work

Pantheon benefits from a decades-long body of related work in Internet measurement, network emulation, transport protocols, and congestion-control schemes.

Tools for Internet measurement. Systems like PlanetLab [10], Emulab [40], and ORBIT [30] provide measurement nodes for researchers to test transport protocols and other end-to-end applications. PlanetLab, which was in wide use from 2004–2012, at its peak included hundreds of nodes, largely on well-provisioned (wired) academic networks around the world. Emulab allows researchers to run experiments over configurable network emulators and on Wi-Fi links within an office building.

While these systems are focused on allowing researchers to borrow nodes and run their own tests, the Pantheon operates at a higher level of abstraction. Pantheon includes a *single* community software package that researchers can contribute algorithms to. Anybody can run any of the algorithms in this package, including over Emulab or any network path, but Pantheon also hosts a common repository of test results (including raw packet traces) of scripted comparative tests.

Network emulation. Congestion-control research has long used network simulators, e.g., ns-2 [28], as well as real-time emulators such as Dummynet [6, 33], NetEm [20], Mininet [19], and Mahimahi [27].

These emulators provide increasing numbers of parameters and mechanisms to recreate different network behaviors, such as traffic shapers, policers, queue disciplines, stochastic i.i.d. or autocorrelated loss, reordering, bit errors, and MAC dynamics. However, properly setting these parameters to emulate a particular target network remains an open problem.

One line of work has focused on improving emulator precision in terms of the level of detail and fidelity at modeling small-scale effects (e.g., "Two aspects influence the accuracy of an emulator: how detailed is the model of the system, and how closely the hardware and software can reproduce the timing computed by the model" [6]). Pantheon takes a different approach, instead focusing on accuracy in terms of how well an emulator recreates the performance of a set of transport algorithms.

Congestion control. Internet congestion control has a deep literature. The original DECBit [32] and Tahoe [22] algorithms responded to one-bit feedback from the network, increasing and decreasing a congestion window in response to acknowledgments and losses. More recently, researchers have tried to formalize the protocol-design process by generating a congestion-control scheme as a function of an objective function and prior beliefs about the network and workload. Remy [37, 41] and PCC [12] are different kinds of "learned" schemes [35]. Remy uses an offline optimizer that generates a decision tree to optimize a global utility score based on network simulations. PCC uses an online optimizer that adapts its sending rate to maximize a local utility score in response to packet losses and RTT samples. In our current work (§ 6), we ask whether it is possible to quickly train an algorithm from first principles to produce good *global* performance on real Internet paths.

3 Design and implementation

This section describes the design and implementation of the Pantheon, a system that automatically measures the performance of many transport protocols and congestion-control schemes across a diverse set of network paths. By allowing the community to repeatably evaluate transport algorithms in scripted comparative tests across real-world network paths, posted to a public archive of results, the Pantheon aims to help researchers develop and test algorithms more rapidly and reproducibly.

Below, we demonstrate several uses for Pantheon: comparing existing congestion-control schemes on real-world networks (§4); calibrating network emulators that accurately reproduce real-world performance (§5); and designing and testing new congestion-control schemes (§6).

Label	Scheme	LoC
Copa	Copa [2]	46
LEDBAT	LEDBAT/μTP [36] (`libutp`)	48
PCC	PCC† [12]	46
QUIC	QUIC Cubic [24] (`proto-quic`)	119
SCReAM	SCReAM [23]	541
Sprout	Sprout† [42]	46
Tao	RemyCC "100x" (2014) [37]	43
BBR	TCP BBR [7]	52
Cubic	TCP Cubic [18] (Linux default)	30
Vegas	TCP Vegas [5]	50
Verus	Verus† [43]	43
—	Vivace [13]	37
WebRTC	WebRTC media [4] in Chromium	283
—	FillP (work in progress)	41
Indigo	LSTM neural network (work in progress)	35

Figure 1: The Pantheon's transport schemes (§3.1.1) and the labels used for them in figures in this paper. Shown are the number of lines of Python, C++, or Javascript code in each wrapper that implements the common abstraction. Schemes marked † are modified to reduce MTU.

3.1 Design overview

Pantheon has three components: (1) a software repository containing pointers to transport-protocol implementations, each wrapped to expose a common testing interface based on the abstraction of a full-throttle flow; (2) testing infrastructure that runs transport protocols in scripted scenarios, instruments the network to log when each packet was sent and received, and allows flows to be initiated by nodes behind a network address translator (NAT); and (3) a global observatory of network nodes, enabling measurements across a wide variety of paths. We describe each in turn.

3.1.1 A collection of transport algorithms, each exposing the same interface

To test each transport protocol or congestion-control scheme on equal footing, Pantheon requires it to expose a common abstraction for testing: a full-throttle flow that runs until a sender process is killed. The simplicity of this interface has allowed us (and a few external contributors so far) to write simple wrappers for a variety of schemes and contribute them to the Pantheon, but limits the kinds of evaluations the system can do.[1]

Figure 1 lists the currently supported schemes, plus the size (in lines of code) of a wrapper script to expose the required abstraction. For all but three schemes, no modification was required to the existing implementation. The remaining three had a hard-coded MTU size and

[1] For example, the interface allows measurements of combinations of long-running flows (with timed events to start and stop a flow), but does not allow the caller to run a scheme until it has transferred exactly *x* bytes. This means that the Pantheon cannot reliably measure the flow-completion time of a mix of small file transfers.

required a small patch to adjust it for compatibility with our network instrumentation; please see §3.1.2 below.

As an example, we describe the Pantheon's wrapper to make WebRTC expose the interface of a full-throttle flow. The Pantheon tests the Chromium implementation of WebRTC media transfer [4] to retrieve and play a video file. The wrapper starts a Chromium process for the sender and receiver, inside a virtual X frame buffer, and provides a signaling server to mediate the initial connection. This comprises about 200 lines of JavaScript.

Pantheon is designed to be easily extended; researchers can add a new scheme by submitting a pull request that adds a submodule reference to their implementation and the necessary wrapper script. Pantheon uses a continuous-integration system to verify that each proposed scheme builds and runs in emulation.

3.1.2 Instrumenting network paths

For each IP datagram sent by the scheme, Pantheon's instrumentation tracks the size, time sent, and (if applicable) time received. Pantheon allows either side (sender or receiver) to initiate the connection, even if one of them is behind a NAT, and prevents schemes from communicating with nodes other than the sender and receiver. To achieve this, Pantheon creates a virtual private network (VPN) between the endpoints, called a *Pantheon-tunnel*, and runs all traffic over this VPN.

Pantheon-tunnel comprises software controlling a virtual network device (TUN) [39] at each endpoint. The software captures all IP datagrams sent to the local TUN, assigns each a unique identifier (UID), and logs the UID and a timestamp. It then encapsulates the packet and its UID in a UDP datagram, which it transmits to the other endpoint via the path under test. The receiving endpoint decapsulates, records the UID and arrival time, and delivers the packet to its own Pantheon-tunnel TUN device.

This arrangement has two main advantages. First, UIDs make it possible to unambiguously log information about every packet (e.g., even if packets are retransmitted or contain identical payloads). Second, either network endpoint can be the sender or receiver of an instrumented network flow over an established Pantheon-tunnel, even if it is behind a NAT (as long as one endpoint has a routable IP address to establish the tunnel).

Pantheon-tunnel also has disadvantages. First, encapsulation costs 36 bytes (for the UID and headers), reducing the MTU of the virtual interface compared to the path under test; for schemes that assume a fixed MTU, Pantheon patches the scheme accordingly. Second, because each endpoint records a timestamp to measure the send and receive time of each datagram, accurate timing requires the endpoints' clocks to be synchronized; endpoints use NTP [29] for this purpose. Finally, Pantheon-tunnel makes all traffic appear to the network as UDP, meaning it cannot measure the effect of a network's discrimination based on the IP protocol type.[2]

Evaluation of Pantheon-tunnel. To verify that Pantheon-tunnel does not substantially alter the performance of transport protocols, we ran a calibration experiment to measure the tunnel's effect on the performance of three TCP schemes (Cubic, Vegas, and BBR). We ran each scheme 50 times inside and outside the tunnel for 30 seconds each time, between a colocation facility in India and the EC2 India datacenter, measuring the mean throughput and 95th-percentile per-packet one-way delay of each run.[3] We ran a two-sample Kolmogorov-Smirnov test for each pair of statistics (the 50 runs inside vs. outside the tunnel for each scheme's throughput and delay). No test found a statistically significant difference below $p < 0.2$.

3.1.3 A testbed of nodes on interesting networks

We deployed observation nodes in countries around the world, including cellular (LTE/UMTS) networks in Stanford (USA), Guadalajara (Mexico), São Paulo (Brazil), Bogotá (Colombia), New Delhi (India), and Beijing (China), wired networks in all of the above locations as well as London (U.K.), Iowa (U.S.), Tokyo (Japan), and Sydney (Australia), and a Wi-Fi mesh network in Nepal. These nodes were provided by a commercial colocation facility (Mexico, Brazil, Colombia, India), by volunteers (China and Nepal), or by Google Compute Engine (U.K., U.S., Tokyo, Sydney).

We found that hiring a commercial colocation operator to maintain LTE service in far-flung locations has been an economical and practical approach; the company maintains, debugs, and "tops up" local cellular service in each location in a way that would otherwise be impractical for a university research group. However, this approach limits us to available colocation sites and ones where we receive volunteered nodes. We are currently bringing up a volunteered node with cellular connectivity in Saudi Arabia and welcome further contributions.

3.2 Operation and testing methods

The Pantheon frequently benchmarks its stable of congestion-control schemes over each path to create an archive of real-world network observations. On each path, Pantheon runs multiple benchmarks per week. Each benchmark follows a software-defined scripted workload (e.g., a single flow for 30 seconds; or multiple flows of

[2] Large-scale measurements by Google [24] have found such discrimination, after deployment of the QUIC UDP protocol, to be rare.

[3] For BBR running outside the tunnel, we were only able to measure the average throughput (not delay). Run natively, BBR's performance relies on TCP segmentation offloading [9], which prevents a precise measurement of per-packet delay without the tunnel's encapsulation.

cross traffic, arriving and departing at staggered times), and for each benchmark, Pantheon chooses a random ordering of congestion-control schemes, then tests each scheme in round-robin fashion, repeating until every scheme has been tested 10 times (or 3 for partly-cellular paths). This approach mirrors the evaluation methods of prior academic work ([12, 42, 43]).

During an experiment, both sides of a path repeatedly measure their clock offset to a common NTP server and use these to calculate a corrected one-way delay of each packet. After running an experiment, a node calculates summary statistics (e.g., mean throughput, loss rate, and 95th-percentile one-way delay for each scheme) and uploads its logs (packet traces, analyses, and plots) to AWS S3 and the Pantheon website (https://pantheon.stanford.edu).

4 Findings

The Pantheon has collected and published measurements of a dozen protocols taken over the course of more than a year. In this section, we give a high-level overview of some key findings in this data, focusing on the implications for research and experimental methodology. We examine comparative performance between protocols rather than the detailed behavior of particular protocols, because comparative analyses provide insight into which protocol end hosts should run in a particular setting.

To ground our findings in examples from concrete data, we select one particular path: AWS Brazil to Colombia. This path represents the performance a device in Colombia would see downloading data from properly geo-replicated applications running in AWS (Brazil is the closest site).

Finding 1: Which protocol performs best varies by path. Figure 2a shows the throughput and delay of 12 transport protocols from AWS Brazil to a server in Colombia, with an LTE modem from a local carrier (Claro).[4] Figure 2b shows the throughput and delay for the same protocols from a node at Stanford University with a T-Mobile LTE modem, to a node in AWS California. The observed performance varies significantly. In Brazil-Colombia, PCC is within 80% of the best observed throughput (QUIC) but with delay 20 times higher than the lowest (SCReAM). In contrast, for Stanford-California, PCC has only 52% of the best observed throughput (Cubic) and the lowest delay. The Sprout scheme, developed by one of the present authors, was designed for cellular networks in the U.S. and performs well in that setting (Figure 2b), but poorly on other paths.

These differences are not only due to long haul paths or geographic distance. Figure 2c shows the performance of the transport protocols from AWS Brazil to a wired device in Colombia. Performance is completely different. Delays, rather than varying by orders of magnitude, differ by at most 32%. At the same time, some protocols are strictly better: QUIC (Cubic) and (TCP) Cubic have both higher throughput and lower delay than BBR and Verus.

Differences are not limited to paths with cellular links. Figure 2e shows performance between Stanford and AWS California using high-bandwidth wired links and Figure 2f shows performance between the Google Tokyo and Sydney datacenters. While in both cases PCC shows high throughput and delay, in the AWS case BBR is better in throughput while between Google data centers it provides 34% less throughput. Furthermore, LEDBAT performs reasonably well on AWS, but has extremely low throughput between Google datacenters.

This suggests that evaluating performance on a small selection (or, in the worst case, just one) of paths can lead to misleadingly positive results, because they are not generalizable to a wide range of paths.

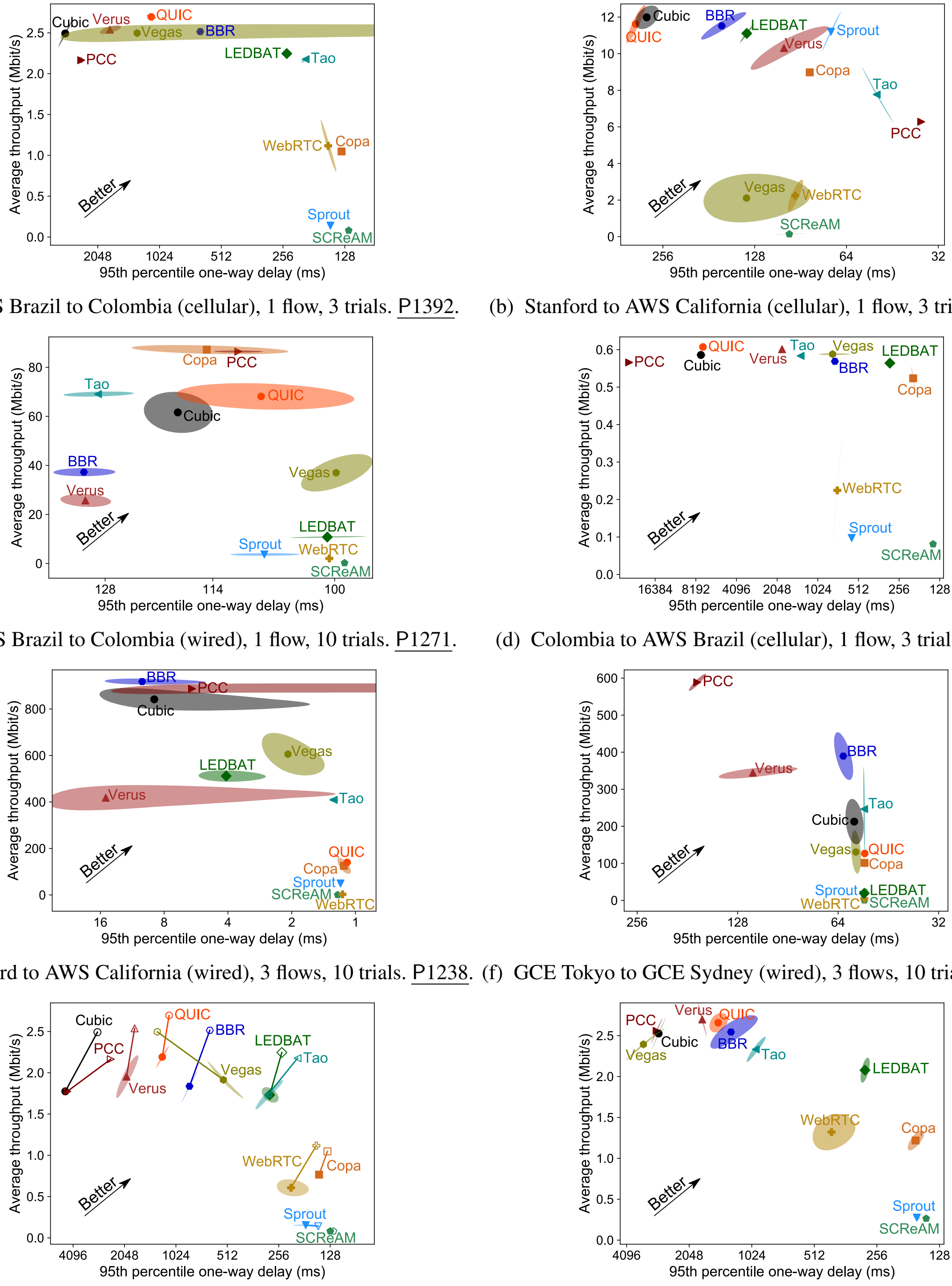
Finding 2: Which protocol performs best varies by path direction. Figure 2d shows the performance of the opposite direction of the path, from the same device with cellular connection in Colombia to AWS Brazil. This configuration captures the observed performance of uploading a photo or streaming video through a relay.

In the Brazil to Colombia direction, QUIC strictly dominates Vegas, providing both higher throughput and lower delay. In the opposite direction, however, the tradeoff is less clear: Vegas provides slightly lower throughput with a significant (factor of 9) decrease in delay. Similarly, in the Brazil to Colombia direction, WebRTC provides about half the throughput of LEDBAT while also halving delay; in the Colombia to Brazil direction, WebRTC is strictly worse, providing one third the throughput while quadrupling delay.

This indicates that evaluations of network transport protocols need to explicitly measure both directions of a path. On the plus side, a single path can provide two different sets of conditions when considering whether results generalize.

Finding 3: Protocol performance varies in time and only slightly based on competing flows. Figure 2g shows the Brazil-Colombia path measured twice, separated by two days (the first measurement shown in open dots is the same as in Figure 2a). Most protocols see a strict degradation of performance in the second measurement, exhibiting lower throughput and higher delay. Cubic and PCC, once clearly distinguishable, merge to have equivalent performance. More interestingly, the performance of Vegas has 23% lower throughput, but cuts delay by more than a factor of 2.

[4] All results in this paper and supporting raw data can be found in the Pantheon archive; e.g. the experiment indicated as P123 can be found at https://pantheon.stanford.edu/result/123/.



(a) AWS Brazil to Colombia (cellular), 1 flow, 3 trials. P1392.

(b) Stanford to AWS California (cellular), 1 flow, 3 trials. P950.

(c) AWS Brazil to Colombia (wired), 1 flow, 10 trials. P1271.

(d) Colombia to AWS Brazil (cellular), 1 flow, 3 trials. P1391.

(e) Stanford to AWS California (wired), 3 flows, 10 trials. P1238.

(f) GCE Tokyo to GCE Sydney (wired), 3 flows, 10 trials. P1442.

(g) AWS Brazil to Colombia (cellular), 1 flow, 3 trials. 2 days after Figure 2a (shown in open dots). P1473.

(h) AWS Brazil to Colombia (cellular), 3 flows, 3 trials. P1405.

Figure 2: Compared with Figure 2a, scheme performance varies across the type of network path (Figure 2c), number of flows (Figure 2h), time (Figure 2g), data flow direction (Figure 2d), and location (Figure 2b). Figure 2e and 2f show that the variation is not limited to just cellular paths. The shaded ellipse around a scheme's dot represents the 1-σ variation across runs. Given a measurement ID, e.g. P123, the full result can be found at https://pantheon.stanford.edu/result/123/.

Finally, Figure 2h shows performance on the Brazil-Colombia path when 3 flows compete. Unlike in Figure 2a, PCC and Cubic dominate Vegas, and many protocols see similar throughput but at greatly increased latency (perhaps due to larger queue occupancy along the path).

This indicates that evaluations of network transport protocols need to not only measure a variety of paths, but also spread those measurements out in time. Furthermore, if one protocol is measured again, all of them need to be measured again for a fair comparison, as conditions may have changed. Cross traffic (competing flows) is an important consideration, but empirically has only a modest effect on relative performance. We do find that schemes that diverge significantly from traditional congestion control (e.g., PCC) exhibit poor fairness in some settings; in a set of experiments between Tokyo and Sydney (P1442), we observed the throughput ratios of three PCC flows to be 32:4:1. This seems to contradict fairness findings in the PCC paper and emphasizes the need for a shared evaluation platform across diverse paths.

5 Calibrated emulators

The results in Section 4 show that transport performance varies significantly over many characteristics, including time. This produces a challenge for protocol development and the ability of researchers to reproduce each others' results. One time-honored way to achieve controlled, reproducible results, at the cost of some realism, is to measure protocols in simulation or emulation [14] instead of the wild Internet, using tools like Dummynet [6, 33], NetEm [20], Mininet [19], or Mahimahi [27].

These tools each provide a number of parameters and mechanisms to recreate different network behaviors, and there is a traditional view in computer networking that the more fine-grained and detailed an emulator, the better. The choice of parameter values to faithfully emulate a particular target network remains an open problem.

In this paper, we propose a new figure of merit for network emulators: the degree to which an emulator can be substituted for the real network path in a full system, including the endpoint algorithm, without altering the system's overall performance. In particular, we define the emulator's accuracy as the average difference of the throughput and of the delay of a set of transport algorithms run over the emulator, compared with the same statistics from the real network path that is the emulator's target. The broader and more diverse the set of transport algorithms, the better characterized the emulator's accuracy will be: each new algorithm serves as a novel probe that could put the network into an edge case or unusual state that exercises the emulator and finds a mismatch.

In contrast to some conventional wisdom, we do not think that more-detailed network models are necessarily preferable. Our view is that this is an empirical question, and that more highly-parameterized network models create a risk of overfitting—but may be justified if lower-parameter models cannot achieve sufficient accuracy.

5.1 Emulator characteristics

We found that a five-parameter network model is sufficient to produce emulators that approximate a diverse variety of real paths, matching the throughput and delay of a range of algorithms to within 17% on average. The resulting calibrated emulators allow researchers to test experimental new schemes—thousands of parallel variants if necessary—in emulated environments that stand a good chance of predicting future real-world behavior.[5]

The five parameters are:

1. a bottleneck link rate,
2. a constant propagation delay,
3. a DropTail threshold for the sender's queue,
4. a stochastic loss rate (per-packet, i.i.d.), and
5. a bit that selects whether the link runs isochronously (all interarrival times equal), or with packet deliveries governed by a memoryless Poisson point process, characteristic of the observed behavior of some LTE networks [42].

To build emulators using these parameters, the Pantheon uses Mahimahi container-based network emulators [27]. In brief: Mahimahi gives the sender and receiver each its own isolated Linux network namespace, or container, on one host. An emulator is defined by a chain of nested elements, each one modeling a specific network effect: e.g., an `mm-loss` container randomly drops packets in the outgoing or incoming direction at a specified rate.

5.2 Automatically calibrating emulators to match a network path

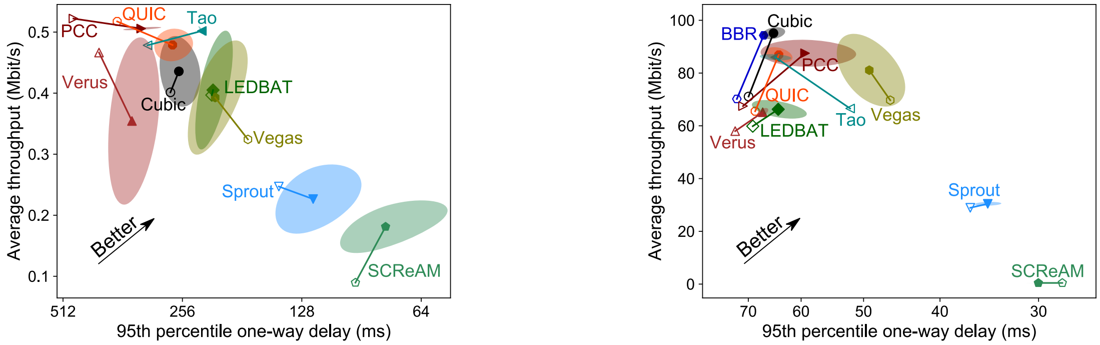
Given a set of results over a particular network path, Pantheon can generate an emulator that replicates the same results in about two hours, using an automated parameter-search process that we now describe.

To find an appropriate combination of emulator parameters, Pantheon searches the space using a non-linear optimization process that aims to find the optimal value for a vector x, which represents the <*rate*, *propagation delay*, *queue size*, *loss rate*> for the emulator.[6]

The optimization derives a replication error for each set of emulator parameters, $f(x)$, which is defined as the

[5] In a leave-one-out cross-validation experiment, we confirmed that emulators trained to match the performance of $n-1$ transport algorithms accurately predicted the unseen scheme's performance within about 20% (results not shown).

[6] The optimization is run twice, to choose between a constant rate or a Poisson delivery process.



(a) Nepal to AWS India (wireless), 1 flow, 10 trials. Mean replication error: 19.1%. P188.

(b) AWS California to Mexico (wired), 3 flows, 10 trials. Mean replication error: 14.4%. P1237.

Figure 3: Examples of per-scheme calibrated emulator errors. The filled dots represent real results over each network path; the open dots represent the corresponding result over the emulator that best replicates all of the results. Emulators for all-wired paths give better fidelity than emulators for partly-wireless paths (§5.3).

Path	Error (%)
Nepal to AWS India (Wi-Fi, 1 flow, P188)	19.1
AWS Brazil to Colombia (cellular, 1 flow, P339)	13.0
Mexico to AWS California (cellular, 1 flow, P196)	25.1
AWS Korea to China (wired, 1 flow, P361)	17.7
India to AWS India (wired, 1 flow, P251)	15.6
AWS California to Mexico (wired, 1 flow, P353)	12.7
AWS California to Mexico (wired, 3 flows, P1237)	14.4

Figure 4: Replication error of calibrated emulators on six paths with a single flow, and one path with three flows of staggered cross traffic.

Path	Feature change	Error (%)
China wired	link rate only	211.8
	add delay	211.8 → 189.7
	add buffer size	189.7 → 32.3
	add stochastic loss	32.3 → 17.7
Colombia cellular	constant → Poisson	23.7 → 13.0

Figure 5: Each of the emulator's five parameters is helpful in reducing replication error. For the China wired path, we started with a single parameter and added the other three features one by one, in the order of their contribution. The Colombia cellular path required jitter (Poisson deliveries) to achieve good accuracy.

average of the percentage changes between the real and emulated mean throughput, and the real and emulated mean 95th-percentile delay, across each of the set of reference transport algorithms. To minimize $f(x)$, nonlinear optimization is necessary because neither the mathematical expression nor the derivative of $f(x)$ is known. In addition, for both emulated and real world network paths, $f(x)$ is non-deterministic and noisy.

The Pantheon uses Bayesian optimization [25], a standard method designed for optimizing the output of a noisy function when observations are expensive to obtain and derivatives are not available.[7] The method starts with the assumption that the objective function, $f(x)$, is drawn from a broad prior (Gaussian is a standard choice and the one we use). Each sample (i.e., calculation of the emulator replication error for a given set of emulator parameters x) updates the posterior distribution for $f(x)$. Bayesian optimization uses an *acquisition function* to guide the algorithm's search of the input space to the next value x. We use the Spearmint [38] Bayesian-optimization toolkit, which uses "expected improvement" as its acquisition function. This function aims to maximize the expected improvement over the current best value [25].

5.3 Emulation results

We trained emulators that model six of Pantheon's paths, each for about 2 hours on 30 EC2 machines with 4 vCPUs each. Figure 3 shows per-scheme calibration results for two representative network paths, a wireless device in Nepal and a wired device in Mexico. Filled dots represent the measured *mean* performance of the scheme on the real network path, while the open dot represents the performance on the corresponding calibrated emulator. A closer dot means the emulator is better at replicating that scheme's performance.

We observe that the emulators roughly preserve the relative order of the mean performance of the schemes on each path. Figure 4 shows mean error in replicating the

[7] Each evaluation of $f(x)$ involves running all of Pantheon's congestion-control schemes in a scripted 30-second scenario, three times, across the emulated path. This is done in parallel, so each evaluation of $f(x)$ takes about 30 seconds of wall-clock time.

throughput and delay performance of all of Pantheon's congestion-control schemes by a series of emulators. To ensure each parameter is necessary, we measured the benefits of adding delay, queue size, and loss information to a base emulator that uses a constant rate, in replicating the China wired device path. For the cellular device path we measured the benefit of using a Poisson based link rate rather than a constant rate. As shown in Figure 5, each added parameter improves the emulator's fidelity.

Pantheon includes several calibrated emulators, and regularly runs the transport algorithms in single- and multi-flow scenarios over each of the emulators and publishes the results in its public archive. Researchers are also able to run the calibrated emulators locally.

In addition, Pantheon includes, and regularly evaluates schemes over, a set of "pathological" emulators suggested by colleagues at Google. These model extreme network behaviors seen in the deployment of the BBR scheme: very small buffer sizes, severe ACK aggregation, and token-bucket policers.

Overall, our intention is that Pantheon will contain a sufficient library of well-understood network emulators so that researchers can make appreciable progress evaluating schemes (perhaps thousands of variants at once) in emulation—with some hope that there will be fewer surprises when a scheme is evaluated over the real Internet.

6 Pantheon use cases

We envision Pantheon as a common evaluation platform and an aid to the development of new transport protocols and congestion-control schemes. In this section, we describe three different ways that Pantheon has been helpful. Two are based on experiences that other research groups have had using Pantheon to assist their efforts. The third is an example of a radical, data-driven congestion-control design based on neural networks learned directly from Pantheon's network emulators.

Case 1. Vivace: validating a new scheme in the real world. Dong et al. [13] describe a new congestion-control scheme called Vivace, the successor to PCC [12]. They contributed three variants of the scheme to Pantheon in order to evaluate and tune Vivace's performance, by examining Pantheon's packet traces and analyses of Vivace in comparison with other schemes across an array of real-world paths. This is consistent with Pantheon's goal of being a resource for the research community (§1).

Case 2. Copa: iterative design with measurements. Arun and Balakrishnan [2] describe another new scheme, Copa, which optimizes an objective function via congestion window and sending rate adjustments. In contrast to Vivace, which was deployed on Pantheon largely as a completed design, Copa used Pantheon as an integral part of the design process: the authors deployed a series of six prototypes, using Pantheon's measurements to inform each iteration. This demonstrates another use of Pantheon, automatically deploying and testing prototypes on the real Internet, and gathering *in vivo* performance data.

Case 3. Indigo: extracting an algorithm from data. As an extreme example of data-driven design, we present Indigo, a machine-learned congestion-control scheme whose design we extract from data gathered by Pantheon.

Using machine learning to train a congestion-control scheme for the real world is challenging. The main reason is that it is impractical to learn directly from the Internet: machine-learning algorithms often require thousands of iterations and hours to weeks of training time, meaning that paths evolve in time (§4) more quickly than the learning algorithm can converge. Pantheon's calibrated emulators (§5) provide an alternative: they are reproducible, can be instantiated many times in parallel, and are designed to replicate the behavior of congestion-control schemes. Thus, our high-level strategy is to train Indigo using emulators, then evaluate it in the real world using Pantheon.

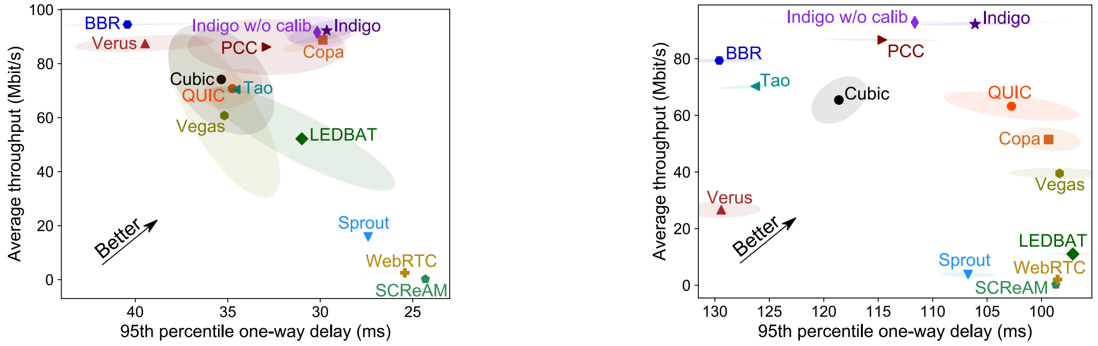
Indigo is one example of what we believe to be a novel family of data-driven algorithms enabled by Pantheon. Specifically, Pantheon facilitates realistic offline training and testing by providing a communal benchmark, evolving dataset, and calibrated emulators to allow approximately realistic offline training and testing. Below, we briefly describe Indigo's design; we leave a more detailed description to future work.

Overview of Indigo

At its core, Indigo does two things: it observes the network state, and it adjusts its *congestion window*, i.e., the allowable number of in-flight packets. Observations occur each time an ACK is received, and their effect is to update Indigo's internal *state*, defined below. Indigo adjusts its congestion window every 10 ms. The state vector is:

1. An exponentially-weighted moving average (EWMA) of the queuing delay, measured as the difference between the current RTT and the minimum RTT observed during the current connection.
2. An EWMA of the sending rate, defined as the number of bytes sent since the last ACK'ed packet was sent, divided by the RTT.
3. An EWMA of the receiving rate, defined as the number of bytes received since the ACK preceding the transmission of the most recently ACK'ed packet, divided by the corresponding duration (similar to and inspired by TCP BBR's delivery rate [7]).
4. The current congestion window size.
5. The previous action taken.

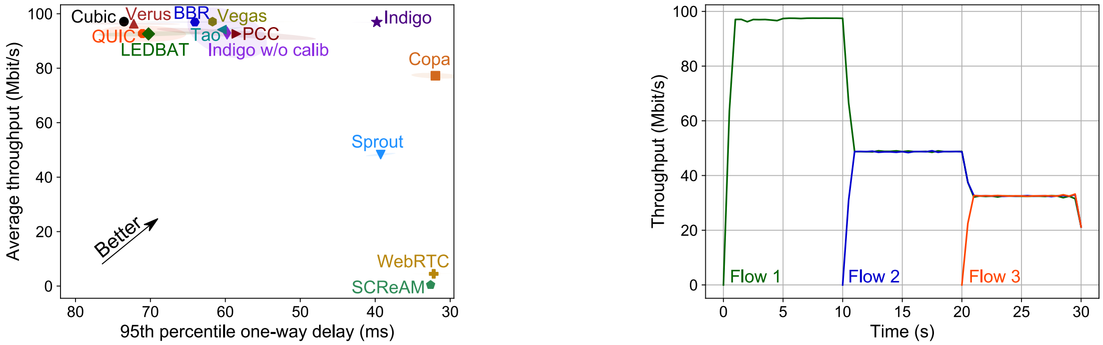
Indigo stores the mapping from states to actions in a Long Short-Term Memory (LSTM) recurrent neural



(a) Mexico to AWS California, 10 trials. P1272.

(b) AWS Brazil to Colombia, 10 trials. P1439.

Figure 6: Real wired paths, single flow. Indigo's performance is at the throughput/delay tradeoff frontier. Indigo without calibrated emulators ("Indigo w/o calib") gives worse performance.



(a) India to AWS India, 10 trials. P1476.

(b) Time-domain three-flow test. One trial in Figure 7a.

Figure 7: Real wired paths, multiple flows. Figure 7a shows the performance of all congestion-control schemes on multi-flow case. Figure 7b shows throughput vs. time for a three-flow run in Figure 7a starting 10 seconds apart. Indigo shares the bandwidth fairly.

network [21] with 1 layer of 32 hidden units (values chosen after extensive evaluation on the Pantheon). Indigo requires a *training phase* (described below) in which, roughly speaking, it learns a mapping from states to actions. Once trained and deployed, this mapping is fixed.

We note that there may be better parameter choices: number of hidden units, action space, state contents, etc. We have found that the above choices already achieve good performance; further improvements are future work. As one step toward validating our choices, we trained and tested several versions of Indigo with a range of hidden units, from 1 to 256, on an emulated network; choices between 16 and 128 yielded good performance.

Indigo's training phase. Indigo uses imitation learning [3, 34] to train its neural network. At a high level, this happens in two steps: first, we generate one or more *congestion-control oracles*, idealized algorithms that perfectly map states to correct actions, corresponding to links on which Indigo is to be trained. Then we apply a standard imitation learning algorithm that use these oracles to generate training data.

Of course, no oracle exists for real-world paths. Instead, we generate oracles corresponding to *emulated* paths; this is possible because Pantheon's emulators (§5) have few parameters. By the definition of an oracle, if we know the ideal congestion window for a given link, we have the oracle for the link: for any state, output whichever action results in a congestion window closest to the ideal value.

A key insight is that for emulated links, we can very closely approximate the ideal congestion window. For simple links with a fixed bandwidth and minimum one-way delay, the ideal window is given by the link's bandwidth-delay product (BDP) per flow. For calibrated emulators (which have DropTail queues, losses, etc.), we compute the BDP and then search near this value in emulation to find the best fixed congestion window size.

After generating congestion-control oracles corresponding to each training link, we use a state-of-the-art imitation learning algorithm called DAgger [34] to train the neural network. For each training link, DAgger trains Indigo's neural network as follows: first, it allows the neural network to make a sequence of congestion-control decisions on the training link's emulator, recording the state vector that led to each decision. Next, it uses the congestion-control oracle to label the correct action corresponding to each recorded state vector. Finally, it updates the neural network by using the resulting state-action mapping as training data. This process is repeated until further training does not change the neural network.

Indigo's performance. In this section, we compare Indigo's performance with that of other congestion-control schemes, and we evaluate the effect of Pantheon's calibrated emulators on performance, versus only training on fixed-bandwidth, fixed-delay emulators.

We trained Indigo on 24 synthetic emulators uncorrelated to Pantheon's real network paths, and on the calibrated emulators (§5). The synthetic emulators comprise all combinations of (5, 10, 20, 50, 100, and 200 Mbps) link rate and (10, 20, 40, 80 ms) minimum one-way delay, with infinite buffers and no loss.

Indigo on Pantheon. We find that Indigo consistently achieves good performance. Figure 6 compares Indigo to other schemes in single flow on two wired paths. In both cases, Indigo is at the throughput/delay tradeoff frontier.

Figure 7 shows Indigo's performance in the multi-flow case. Figure 7a shows the performance of all of Pantheon's congestion-control schemes on a wired path from India to AWS India; Indigo is once again on the throughput/delay tradeoff frontier. Figure 7b is a time-domain plot of one trial from Figure 7a, suggesting that Indigo shares fairly, at least in some cases.

Benefit of calibrated emulators. Figures 6 and 7 also depict a variant of Indigo, "Indigo w/o calib," that is only trained on the synthetic emulators, but not the calibrated emulators. The version trained on calibrated emulators is always as least as good or better.

7 Discussion, limitations, and future work

Improving Pantheon. Pantheon would be more useful if it collected more data about congestion-control schemes. For instance, Pantheon currently gathers data only from a handful of nodes—vastly smaller than the evaluations large-scale operators can perform on even a small fraction of a billion-user population.

Moreover, geographic locality does not guarantee network path similarity: two nodes in the same city can have dramatically different network connections. Pantheon also only tests congestion-control schemes at full throttle; other traffic patterns (e.g., Web-like workloads) may provide researchers with valuable information (e.g., how their scheme affects page-load times).

Finally, Pantheon currently measures the interaction between multiple flows of cross-traffic governed by the same scheme, but we are working to make it measure interactions between different schemes. These measurements will help evaluate fairness in the real world.

Improving the calibrated emulators. Our current emulators replicate throughput and delay metrics only within 17% accuracy on average. An open question is whether we can improve emulator fidelity—especially on cellular paths—without risk of overfitting. Considering metrics other than 95th-percentile delay and mean throughput may be one path forward. Adding more schemes to Pantheon could also help—or it might reveal that the current set of emulator parameters, which we have empirically determined, is insufficient for some schemes.

Indigo. We have presented a case study on Indigo, a data-driven approach to congestion-control design that crucially relies on Pantheon's family of emulators. Indigo's trained model is complex and may have unknown failure modes, but the results to date demonstrate how Pantheon can enable new approaches to protocol design.

8 Conclusion

The Pantheon is a collection of transport protocols and a distributed system of measurement points and network emulators for evaluating and developing them. By measuring many transport protocols and congestion-control algorithms across a diverse set of paths, Pantheon provides a training ground for studying and improving their performance. Furthermore, by generating calibrated emulators that match real-world paths, Pantheon enables researchers to measure protocols reproducibly and accurately.

Pantheon has assisted in the development of two recently-published congestion-control algorithms [2, 13], and has supported our own data-driven approach to protocol design. In other areas of computer science, community benchmarks and recurrent bakeoffs have fueled advances and motivated researchers to build on each others' work: ImageNet in computer vision, the TPC family and Sort Benchmarks for data processing, Kaggle competitions in machine learning, etc. We are hopeful that Pantheon will, over time, serve a similar role in computer networking.

Acknowledgments

We thank the USENIX ATC reviewers for their helpful comments and suggestions. We are grateful to Yuchung Cheng, Janardhan Iyengar, Michael Schapira, and Hari Balakrishnan for feedback throughout this project. This work was supported by NSF grant CNS-1528197, DARPA grant HR0011-15-2-0047, Intel/NSF grant CPS-Security-1505728, the Secure Internet of Things Project, and by Huawei, VMware, Google, Dropbox, Facebook, and the Stanford Platform Lab.

References

[1] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP (DCTCP). In *ACM SIGCOMM computer communication review* (2010), vol. 40, ACM, pp. 63–74.

[2] ARUN, V., AND BALAKRISHNAN, H. Copa: Practical delay-based congestion control for the internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2018), USENIX Association.

[3] BENGIO, S., VINYALS, O., JAITLY, N., AND SHAZEER, N. Scheduled sampling for sequence prediction with recurrent neural networks. *CoRR abs/1506.03099* (2015).

[4] BERGKVIST, A., BURNETT, D. C., JENNINGS, C., NARAYANAN, A., AND ABOBA, B. Webrtc 1.0: Real-time communication between browsers. *Working draft, W3C 91* (2012).

[5] BRAKMO, L. S., O'MALLEY, S. W., AND PETERSON, L. L. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM* (1994).

[6] CARBONE, M., AND RIZZO, L. Dummynet revisited. *SIGCOMM Comput. Commun. Rev. 40*, 2 (Apr. 2010), 12–20.

[7] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. BBR: Congestion-based congestion control. *Queue 14*, 5 (2016), 50.

[8] CHEN, J., SUBRAMANIAN, L., IYENGAR, J., AND FORD, B. TAQ: Enhancing fairness and performance predictability in small packet regimes. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 7:1–7:14.

[9] CHENG, Y., AND CARDWELL, N. Making Linux TCP fast.

[10] CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M., AND BOWMAN, M. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review 33*, 3 (July 2003), 00–00.

[11] DENG, J., DONG, W., SOCHER, R., LI, L.-J., LI, K., AND FEI-FEI, L. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on* (2009), IEEE, pp. 248–255.

[12] DONG, M., LI, Q., ZARCHY, D., GODFREY, P. B., AND SCHAPIRA, M. PCC: Re-architecting congestion control for consistent high performance. In *Presented as part of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[13] DONG, M., MENG, T., ZARCHY, D., ARSLAN, E., GILAD, Y., GODFREY, B., AND SCHAPIRA, M. PCC Vivace: Online-learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2018), USENIX Association.

[14] FALL, K., AND FLOYD, S. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *SIGCOMM Comput. Commun. Rev. 26*, 3 (July 1996), 5–21.

[15] FLOYD, S., AND KOHLER, E. Internet research needs better models. *SIGCOMM Comput. Commun. Rev. 33*, 1 (Jan. 2003), 29–34.

[16] FLOYD, S., AND PAXSON, V. Difficulties in simulating the internet. *IEEE/ACM Trans. Netw. 9*, 4 (Aug. 2001), 392–403.

[17] GERSHGORN, D. The data that transformed AI research—and possibly the world. Quartz, https://qz.com/1034972/the-data-that-changed-the-direction-of-ai-research-and-possibly-the-world, July 2017.

[18] HA, S., RHEE, I., AND XU, L. CUBIC: A new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev. 42*, 5 (July 2008), 64–74.

[19] HANDIGOL, N., HELLER, B., JEYAKUMAR, V., LANTZ, B., AND MCKEOWN, N. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2012), CoNEXT '12, ACM, pp. 253–264.

[20] HEMMINGER, S. Network emulation with NetEm. In *Linux conf au* (2005), pp. 18–23.

[21] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[22] JACOBSON, V. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols* (New York, NY, USA, 1988), SIGCOMM '88, ACM, pp. 314–329.

[23] JOHANSSON, I., AND SARKER, Z. Self-clocked rate adaptation for multimedia. Tech. Rep. RFC8298, Internet Engineering Task Force, 2017.

[24] LANGLEY, A., RIDDOCH, A., WILK, A., VICENTE, A., KRASIC, C., ZHANG, D., YANG, F., KOURANOV, F., SWETT, I., IYENGAR, J., ET AL. The QUIC transport protocol: Design and Internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 183–196.

[25] MOČKUS, J. On Bayesian methods for seeking the extremum. In *Optimization Techniques IFIP Technical Conference* (1975), Springer, pp. 400–404.

[26] Netflix Open Connect. https://openconnect.netflix.com/.

[27] NETRAVALI, R., SIVARAMAN, A., DAS, S., GOYAL, A., WINSTEIN, K., MICKENS, J., AND BALAKRISHNAN, H. Mahimahi: Accurate record-and-replay for HTTP. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2015), USENIX ATC '15, USENIX Association, pp. 417–429.

[28] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[29] NTP: The network time protocol. http://www.ntp.org/.

[30] OTT, M., SESKAR, I., SIRACCUSA, R., AND SINGH, M. ORBIT testbed software architecture: Supporting experiments as a service. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities* (Washington, DC, USA, 2005), TRIDENTCOM '05, IEEE Computer Society, pp. 136–145.

[31] PAXSON, V., AND FLOYD, S. Why we don't know how to simulate the Internet. In *Proceedings of the 29th Conference on Winter Simulation* (Washington, DC, USA, 1997), WSC '97, IEEE Computer Society, pp. 1037–1044.

[32] RAMAKRISHNAN, K. K., AND JAIN, R. A binary feedback scheme for congestion avoidance in computer networks. *ACM Trans. on Comp. Sys. 8*, 2 (May 1990), 158–181.

[33] Rizzo, L. Dummynet: A simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev. 27*, 1 (Jan. 1997), 31–41.

[34] Ross, S., Gordon, G. J., and Bagnell, J. A. No-regret reductions for imitation learning and structured prediction. *CoRR abs/1011.0686* (2010).

[35] Schapira, M., and Winstein, K. Congestion-control throwdown. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2017), HotNets-XVI, ACM, pp. 122–128.

[36] Shalunov, S., Hazel, G., Iyengar, J., and Kuehlewind, M. Low extra delay background transport (LEDBAT). Tech. Rep. RFC6817, Internet Engineering Task Force, 2012.

[37] Sivaraman, A., Winstein, K., Thaker, P., and Balakrishnan, H. An experimental study of the learnability of congestion control. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 479–490.

[38] Snoek, J., Larochelle, H., and Adams, R. P. Practical Bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems* (2012), pp. 2951–2959.

[39] Universal TUN/TAP device driver. https://www.kernel.org/doc/Documentation/networking/tuntap.txt.

[40] White, B., Lepreau, J., and Guruprasad, S. Lowering the barrier to wireless and mobile experimentation. *SIGCOMM Comput. Commun. Rev. 33*, 1 (Jan. 2003), 47–52.

[41] Winstein, K., and Balakrishnan, H. TCP ex machina: Computer-generated congestion control. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 123–134.

[42] Winstein, K., Sivaraman, A., and Balakrishnan, H. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013), USENIX, pp. 459–471.

[43] Zaki, Y., Pötsch, T., Chen, J., Subramanian, L., and Görg, C. Adaptive congestion control for unpredictable cellular networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 509–522.

ClickNF: a Modular Stack for Custom Network Functions

Massimo Gallo and Rafael Laufer
Nokia Bell Labs

Abstract

Network function virtualization has recently allowed specialized equipment to be replaced with equivalent software implementation. The Click router was a first step in this direction, defining a modular platform for generalized packet processing. Despite its major impact, however, Click does not provides native L4 implementation and only uses nonblocking I/O, limiting its scope to L2-L3 network functions. To overcome these limitations we introduce ClickNF, which provides modular transport and application-layer building blocks for the development of middleboxes and server-side network functions. We evaluate ClickNF to highlight its state-of-the-art performance and showcase its modularity by composing complex functions from simple elements. ClickNF is open source and publicly available.

1 Introduction

Software-defined networking had a significant impact on the packet forwarding infrastructure, providing flexibility and controllability to network and datacenter operators [37]. In a similar trend, network function virtualization (NFV) is sparking novel approaches for deploying flexible network functions [19], ranging from virtual machine orchestration [24, 36, 34] to new packet processing frameworks [8, 40]. Network functions can combine packet forwarding and simple header rewriting with awareness of stateful transport logic, and possibly execute complex application-layer operations.

A modular L2–L7 data plane would offer several advantages for the development of new network functions, such as decoupling state and packet processing, extensibility of fine-grained protocol behavior, module reuse, and a simplification of cross-layer protocol optimizations and debugging. Among existing approaches, Click [29] is arguably the best starting point for such an architecture due to its modularity and extensibility. However, several functionalities are still missing to make Click into a full-stack modular data plane for network functions. First, it lacks L4 native implementation, preventing cross-layer optimizations and stack customization. Second, it has no support for blocking I/O primitives, forcing developers to use more complex asynchronous non-blocking I/O. Third, Click applications must resort to the OS stack, which leads to severe I/O bottlenecks. Finally, despite recent improvements, Click does not support hardware offloading and efficient timer management preventing it to scale at high-speed in particular scenarios.

In this paper we introduce ClickNF, a framework that overcomes the aforementioned limitations and enables L2–L7 modular network function development in Click. Along with legacy Click elements, ClickNF enables developers to overhaul the entire network stack, if desired. First, it introduces a modular TCP implementation that supports options, congestion control, and RTT estimation. Second, it introduces blocking I/O support, providing applications with the illusion of running uninterrupted. Third, it exposes standard socket, zero-copy, and socket multiplexing APIs as well as basic application layer building blocks. Finally, to improve scalability, ClickNF integrates I/O acceleration techniques first introduced in Fastclick [9], such as Data Plane Development Kit (DPDK) [33] and batch processing with additional support for hardware acceleration, as well as an improved timer management system for Click.

ClickNF can be used to deploy a vast class of network functions. For middleboxes, TCP termination is needed for Split TCP, L7 firewalls, TLS/SSL proxies, HTTP caches, etc. At the network edge, ClickNF can be used to implement high-speed modular L7 servers using socket multiplexing primitives to handle I/O events efficiently. As proof of concept, we compose an HTTP cache server with optional SSL/TLS termination and a SOCKS4 proxy. We show that ClickNF provides equivalent performance and scalability as existing user-space stacks while enabling L2–L7 modularity.

The paper is organized as follows. Section 2 describes ClickNF design. Section 3 details our TCP implementation in Click and Section 4 presents application layer modularity. Section 5 highlights a number of original aspects about the ClickNF implementation that are evaluated in Section 6. Section 7 reviews the related work and Section 8 concludes the paper.

2 ClickNF

ClickNF leverages Click [29], a software architecture for building modular and extensible routers. Before introducing its design, we first provide an overview of Click.

2.1 Background

A router in Click is built from a set of fine-grained packet processing modules, *elements*, implementing simple functions (e.g., IP routing). A *configuration* file connects these elements together into a directed graph, whose edges specify the path that packets shall traverse. Depending on the configuration, users can implement network functions of arbitrary complexity (e.g., switch).

Each element may define any number of input and output *ports* to connect to other elements. Ports operate in either *push* or *pull* mode. On a push connection, the packet starts at the source element and moves to the destination element downstream. On a pull connection, in contrast, the destination element requests a packet from the upstream one, which returns a packet if available or a null pointer otherwise. In addition to push or pull, a port may also be *agnostic* and behave as either push or pull depending on the port it is connected to.

In its underlying implementation, Click employs a task queue and a timer priority queue. An infinite loop runs tasks in sequence and timers at expiration. Tasks are element-defined functions that require CPU scheduling, and initiate a sequence of push or pull requests. Most elements, however, do not require their own task, since their `push` and `pull` methods are called by a scheduled task. Timer callback functions are similar to tasks, except for being scheduled at a particular time.

2.2 Design

Network protocol stacks are typically implemented as monolithic packages, either in kernel or user-space. Network function developers often experience hurdles when attempting to debug and customize their software to obtain the desired effects, as the inner workings of the stacks are not exposed. Indeed, recent work [22, 17, 39, 38] advocates that legacy network stacks prevent innovation due to the lack of flexibility and propose to move some of their functionalities outside of the data path.

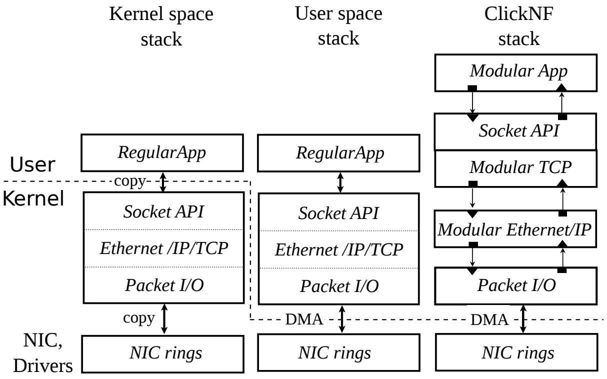


Figure 1: ClickNF design compared to alternatives.

Modular, configurable, and extensible transport protocols were proposed in the past by the research community [16, 13] and by the Linux kernel one [1] constituting a first step in the right direction. Our goal, is along the same lines but broader. ClickNF aims to give developers unfettered access to the entire stack by providing a framework for the construction of modular L2-L7 network functions without the concerns for the correctness of its behavior nor the constraints added by event-driven domain-specific APIs [26].

The design of ClickNF combines the modularity and flexibility of Click with high-speed packet I/O and ready-made protocol building blocks for transport and application-layer features. Figure 1 compares ClickNF design against legacy OSs and user space stacks. In contrast with other approaches that conceal network stack complexity into a monolithic package or does not introduce modularity at all layers, we decompose the full L2–L7 stack into several simple elements that can be individually replaced, modified or removed by rewiring the configuration file, providing a level of flexibility that is not available with alternative solutions. Additionally, elements can be aggregated into a single macro-element to hide complexity when desired. The rationale behind this fine-grained decomposition is twofold. First, simple elements allow the modification and control of each aspect and mechanism of network protocols. This enables module reuse in other contexts, such as recycling existing congestion control strategies to implement new protocols like QUIC [30], or new strategies such as BBR [14] or DCTCP [6] with little effort. Second, this approach helps decoupling protocol state management and packet processing, simplifying complicated tasks such as full state migration between servers or across heterogeneous hardware (e.g., between CPUs and smart NICs).

In the rest of the paper we focus on the description and evaluation of ClickNF transport and application layers, and on some important implementation details that allows ClickNF to sustain line-rate.

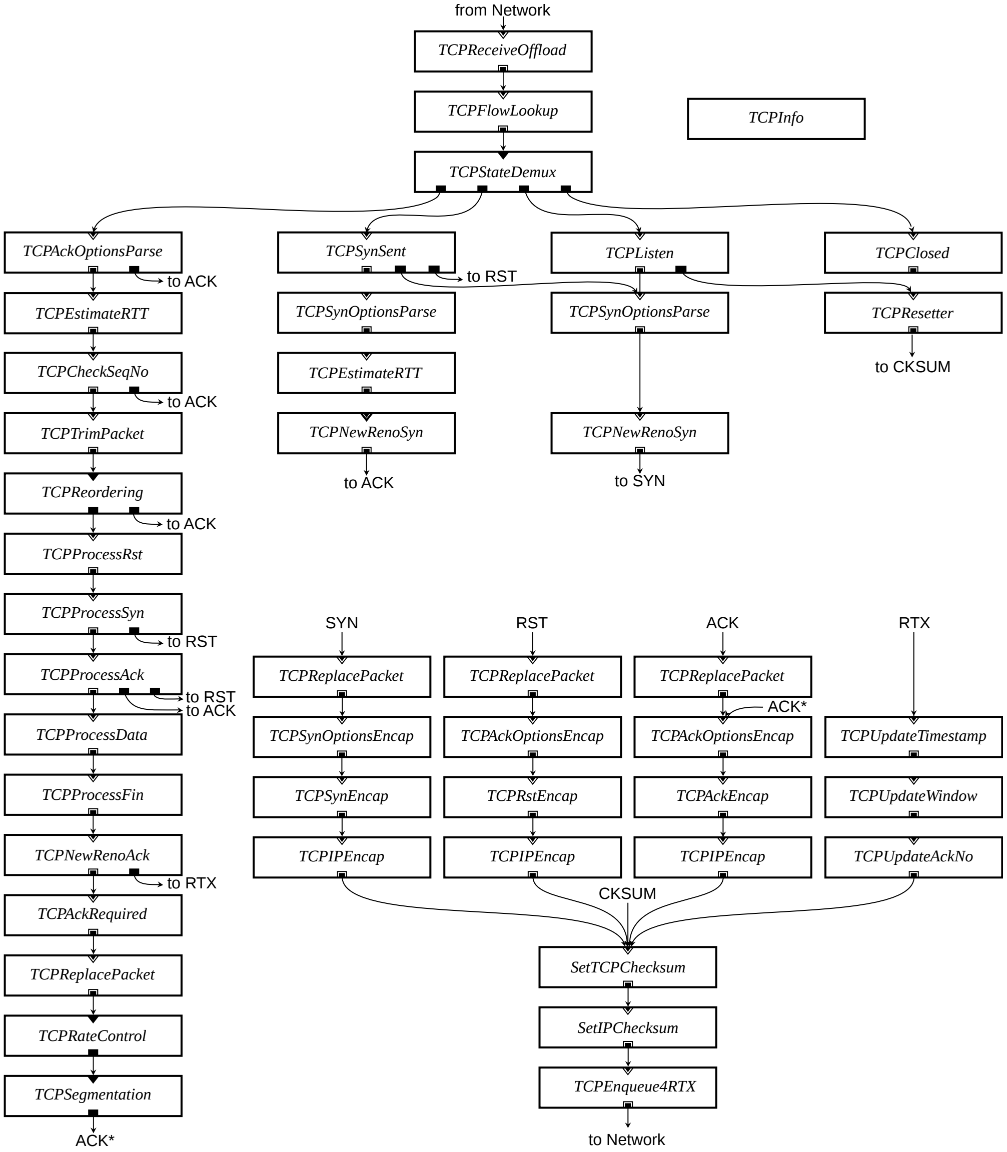


Figure 2: cTCP configuration for incoming network packets.

3 Click TCP

Our modular Click TCP (cTCP) implementation is compliant with IETF standards (RFCs 793 and 1122) and supports TCP options (RFC 7323), New Reno congestion control (RFCs 5681 and 6582), timer management, and RTT estimation (RFC 6298). In this section we describe the cTCP element graphs used to process incoming and outgoing TCP packets.

3.1 Incoming packets

The key element of cTCP is *TCPInfo*, which enables state decoupling by providing other elements with access to important data structures (*i.e.,* TCP Control Block). Figure 2 shows the cTCP element graph for processing incoming packets. In essence, elements access and/or modify the TCP control block (TCB) via the *TCPInfo*, as the packet moves along the edges of the graph. The vertical paths are the directions that most received packets take. The long element sequence on the left represents the packet processing of an established TCP connection. The three other paths to the right take care of special situations, such as connection establishment and termination. Other paths in the graph represent a disruption in the expected packet flow, e.g., *TCPCheckSeqNo* sends an ACK back if the data is outside the receive window.

Most elements in cTCP only require the TCB to process packets. For instance, *TCPTrimPacket* trims off any out-of-window data from the packet. *TCPReordering* enforces in-order delivery by buffering out-of-order packets and releasing them in sequence once the gap is filled.

Elements like *TCPProcess*{*Rst, Syn, Ack, Fin*} inspect the respective TCP flags and react accordingly. In presence of new data, *TCPProcessData* clones the packet (*i.e.,* only packet's metadata are copied) and places it on the RX queue for the application. After receiving three duplicate ACKs, *TCPNewRenoAck* retransmits the first unacknowledged packet. Related elements, such as *TCPNewRenoSyn* and *TCPNewRenoRTX*, handle initialization and retransmissions in congestion control.

In addition to the TCB, other cTCP elements require information previously computed by other elements. This is supported in Click via *packet annotations*, *i.e.,* packet metadata. cTCP packet annotations include:

TCB pointer: The TCB table is stored as a per-core hash table in *TCPInfo* and accessed by other elements using static functions. For each packet, *TCPFlowLookup* looks the TCP flow up in the table and sets the TCB annotation to allow other elements to easily access/modify the TCB avoiding multiple flow table lookups.

RTT measurement: *TCPAckOptionsParse* computes the RTT from the TCP timestamp, Karn's algorithm, and sets it as an annotation. If TCP timestamps are not provided by NICs or by packet I/O elements, *TCPEnqueue4RTX* timestamps each transmitted packet before storing it in the retransmission (RTX) queue. In both cases, *TCPEstimateRTT* uses these annotations to estimate the RTT and update the retransmission timeout.

Acknowledged bytes: *TCPProcessAck* computes the number of acknowledged bytes in each packet and sets it as an annotation. This is later read by *TCPNewRenoAck* to increase the congestion window. If this number is zero and a few other conditions hold (e.g., the receive window is unchanged), the packet is considered a duplicate ACK and may trigger a fast retransmission.

Flags: TCP flags are used to indicate certain actions to other elements. For instance, *TCPProcessData* sets a flag when the received packet has new data. *TCPAckRequired* then checks this flag and, if set, pushes the packet out to trigger an ACK transmission.

3.2 Outgoing packets

Figure 3 shows the cTCP element graph for processing outgoing packets. Applications send data using socket or zero-copy APIs (Section 3.3) that invoke static functions in *TCPSocket*. For each socket call, this element looks up the socket file descriptor in a per-core socket table in *TCPInfo*. For transmissions, *TCPNagle* first checks if the packet should be sent according to Nagle's algorithm. *TCPRateControl* then verifies whether send and congestion windows allow packet transmission. If so, *TCPSegmentation* breaks the data up into MTU-sized packets, and *TCPAckOptionEncap*, *TCPAckEncap*, and *TCPIPEncap* prepend TCP and IP header respectively. Before sending it to lower layers, *TCPEnqueue4RTX* clones the packet and saves it in the retransmission queue until acknowledged by the other end. To initiate a TCP connection, *TCPSynOptionEncap* and *TCPSynEncap* generate the TCP options and header and forward the packet downstream. Similarly, to terminate the connection, *TCPAckOptionEncap* and *TCPFinEncap* form a TCP packet with the FIN flag set.

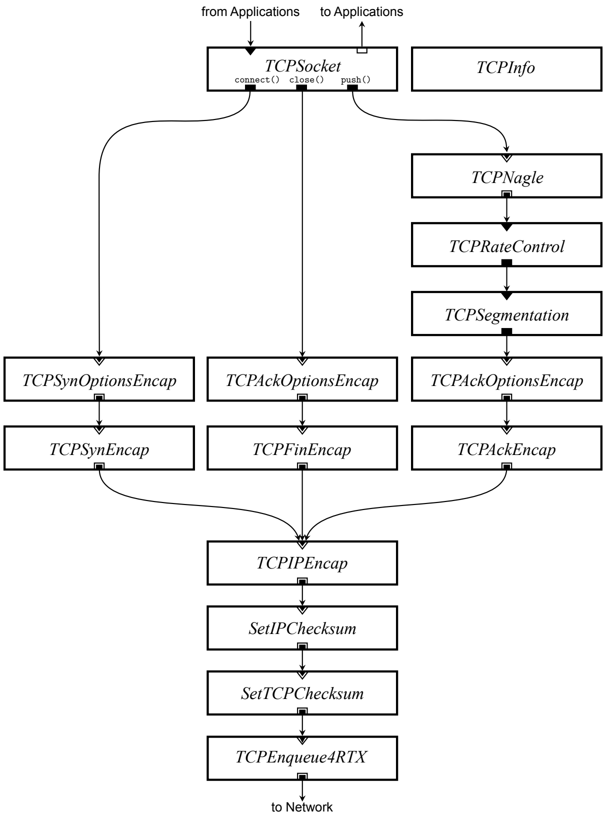


Figure 3: cTCP configuration for outgoing packets.

3.3 Transport APIs

cTCP APIs are designed with two contrasting objectives in mind: (*i*) minimize the efforts required to port application logic in ClickNF; and (*ii*) provide primitives to guarantee high performance at the cost of more complex development. We therefore provide two APIs to interact with the ClickNF transport layer, and one for socket I/O multiplexing.

Socket API: For each socket system call (e.g., `send`), cTCP provides a corresponding function (e.g., `click_send`) for both blocking or non-blocking mode. As in Linux, the operation mode is set on a per-socket basis using the `SOCK_NONBLOCK` flag. In case of blocking sockets, the application is blocked when waiting on I/O; in case of non-blocking sockets, the socket calls return immediately.

Zero-copy interface: In addition to standard Socket API, cTCP provides `click_push` and `click_pull` functions to enable zero-copy operations. For transmissions, applications first allocate a packet and write data to it before pushing it down. cTCP then adds the protocol headers and transmits the packet(s) to the NIC. For receptions, packets are accessed, processed, and placed into the RX queue of applications. To amortize per-packet overhead, both functions can also send and receive batches, and operate in either blocking or non-blocking mode.

Socket I/O multiplexing: To avoid busy-waiting on I/O, cTCP provides a `click_poll` and a `click_epoll_wait` functions to multiplex events from several socket file descriptors. As in the regular epoll API provided by Linux kernel, applications must first register the monitored socket file descriptors with `click_epoll_ctl` and then use `click_epoll_wait` to wait on I/O.

3.4 Timer Management

In Click, timers corresponds to tasks scheduled at a given time in the future. TCP timers are used for retransmissions, keepalive messages, and delayed ACKs. Their implementation in cTCP is critical for performance and scalability reasons. Figure 4 shows cTCP timers' configuration. After a retransmission timeout task is scheduled, *TCPTimer* dequeues the head-of-line packet from the RTX queue and pushes it out. *TCPUpdateTimestamp*, *TCPUpdateWindow*, and *TCPUpdateAckNo* update the respective TCP header fields. Similarly when a keepalive timeout expires, *TCPTimer* pushes an empty packet, and *TCPAckOptionsEncap*, *TCPAckEncap* generate the TCP header of a regular ACK packet. *DecTCPSeqNo* then decrements the TCP sequence number to trigger an ACK back from the other host. Finally, delayed ACKs are sent for every pair of received data packets unless a 500 ms timeout elapses. In this case, a regular ACK is sent using the modules described above.

3.5 Customization and Element Reuse

cTCP modularity enables code reuse and fine-grained customization of the network stack. For instance, TCP reliability can be disabled by simply removing *TCPEnqueue4RTX* from the configuration file. In this section, we present concrete examples to showcase the benefits of our modular TCP implementation.

Building a traffic generator that emulates several TCP flows concurrently sending at a constant rate is straightforward with ClickNF elements. The *TCPInfo* element is inserted in the configuration file and initialized with the corresponding TCBs. Data packets are generated and shaped at a constant rate by regular Click elements, *InfiniteSource* and *Shaper*, and then forwarded to a new element, *TCBSelector*, that randomly associates the packet to an existing TCB using ClickNF annotations. Afterwards, packets go through ClickNF elements such as *TCPAckEncap*, *TCPIPEncap* (plus optionally *SetTCPChecksum*, *SetIPChecksum*) to fill IP and TCP headers before being forwarded to an I/O element.

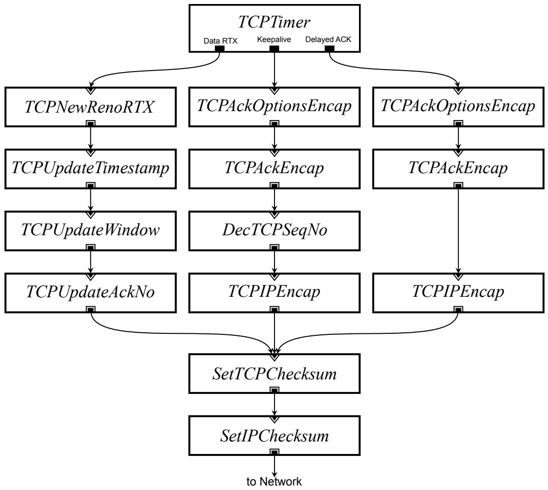


Figure 4: cTCP configuration for TCP timers.

Moreover, per-flow congestion control can be used to ensure that specific traffic classes are processed using appropriate algorithms. Implementing such a feature in a monolithic OS network stack (e.g., Linux kernel one) is, however, quite complicated. Due to its modularity, ClickNF allows the definition of per-flow congestion control by simply inserting a *Classifier* element that modifies the behavior of cTCP for specific flows.

Finally, changing TCP New Reno to match data center TCP (DCTCP) [6] is as simple as adding a new *DCTCP-ProcessECN* element right after *TCPProcessAck* (Figure 2). This element modifies the TCP window behavior in presence of explicit congestion notification. Similarly, the introduction of a new congestion control algorithm, such as BBR [14], requires the development of few additional elements of low complexity.

4 Modular application

ClickNF enables the development of modular applications on top of cTCP. L7 network functions can be implemented using several flow-oriented elements, enabling the programmer to decouple packet processing from application state management logic. To do so, ClickNF separates network and application execution contexts in order to allow applications to block their execution when waiting for I/O operations. ClickNF also provides two fundamental building blocks, *i.e.,* socket I/O multiplexing and SSL/TLS termination elements, that enable rapid composition of complex and customized L7 functions.

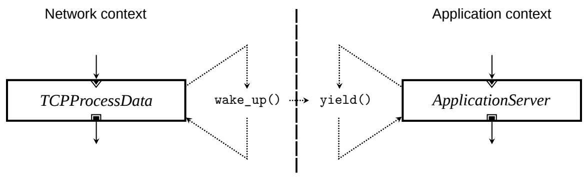


Figure 5: Context switch: *TCPProcessData* reschedules a blocked task waiting on I/O.

4.1 Blocking Tasks

ClickNF implements blocking I/O to provide developers with a broader range of I/O options. In Click, *tasks* are element-defined functions that require frequent CPU scheduling, and initiate a sequence of packet processing in the configuration graph. We introduce the concept of *blocking tasks*, which can yield the CPU if a given condition does not hold, e.g., an I/O request cannot be promptly completed. When rescheduled, the task resumes exactly where it left off, providing applications with an illusion of continuous execution. Blocking tasks are backward compatible with regular tasks, and require no modifications to the Click task scheduler.

Context switching between tasks is light-weight, saving and restoring registers required for task execution. ClickNF uses low-level functions to save and restore the tasks context, just as in POSIX threads. Differently than POSIX threads, however, ClickNF has access to the Click task scheduler and relies on cooperative, as opposed to preemptive, scheduling. ClickNF uses the Boost library to perform context switches in a handful of CPU cycles, *i.e.,* ≈20 cycles in x86_64 (few nanoseconds).

4.2 Network and Application Contexts

ClickNF uses blocking tasks to separate network and application execution contexts. The network context is active during packet reception (cf. Figure 3) and timeouts (cf. Figure 4), and runs through regular Click tasks. The application context, in contrast, is active during application processing and packet transmission (cf. Figure 2) and runs through blocking tasks.

A blocked application is scheduled when the event it is waiting on occurs, e.g., a task blocked on `accept` . Figure 5 presents an example of an application task being rescheduled by an event. In the example, *ApplicationServer* calls `epoll_wait` to monitor a group of active file descriptors. Since no one is ready to perform I/O, it calls `yield` to save the current context and unschedule the task. Later on, when a data packet is received, *TCPProcessData* checks if the application task is waiting for data packets and calls `wake_up` to reschedule it. The event is then inserted into a per-core event queue that stores the events that occurred for the sockets monitored by *ApplicationServer* (*i.e.,*different applications have separate event queues). When the application task is executed, to amortize the cost of the context switch, a batch of events is handled before the network context is re-scheduled.

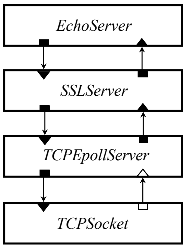


Figure 6: Configuration graph of a modular echo server in ClickNF that uses SSL/TLS encryption.

In ClickNF we specify a list of events needed to manage cTCP states and error conditions. For instance, events are generated when the accept queue becomes non-empty, or when the connection is established to wake up application tasks waiting on these conditions. Similarly, events are also generated when the TX queue becomes non-full, the RX queue becomes non-empty, and also when the RTX queue becomes empty. Other events signal that the remote peer wants to close the connection, the connection was closed, or an error occurred (e.g., a reset or timeout). Despite this fine grain event characterization, to remain compliant with the original epoll API, we map cTCP events to standard `EPOLLIN`, `EPOLLOUT`, and `EPOLLERR` events.

4.3 Application Building Blocks

To simplify application-level programming and promote code reuse, ClickNF provides four building blocks useful for practically relevant network functions. Such building blocks exchange control information with application layer elements using packet annotations. In this way, application elements are informed about the socket file descriptor to which the packets belongs and about new or closed connections. This allows them to efficiently multiplex data between different applications and multiple sockets in both directions.

The first application-layer building block, *TCPEpollServer*, implements an epoll server concealing the complexity of cTCP event handling and can be used to rapidly implement server-side network functions. Similarly, *TCPEpollClient* implements an epoll client to multiplex outgoing connections. Finally, *SSLServer* and *SSLClient* provide SSL/TLS encryption through the OpenSSL library, and may be used to implement network function that require end-to-end encryption. Application data enters in an input port of *SSLServer* as plaintext and leaves its output port as ciphertext; received packets take the reverse path to decrypt ciphertext into plaintext.

Figure 6 shows an echo server using SSL/TLS encryption as a straightforward example of a modular application assembled from ClickNF building blocks. Upon reception at *TCPEpollServer*, packets are decrypted by *SSLServer* and forwarded upstream. *EchoServer* is a stateless application that simply redirects the received data back to the client. On the way back, *SSLServer* encrypts the data and forwards it downstream to *TCPEpollServer*. This simple example shows how ClickNF enables a large number of possibilities for customizing the entire network stack of an application since any of its building blocks can be easily disabled or rewired. For instance, the echo server in Figure 6 can easily disable SSL/TLS for selected flows by introducing a classifier element into the configuration graph, without any change to the element that implements the application logic.

5 Implementation

ClickNF benefits from several improvements that the Click codebase received over time, such as fast packet I/O and multicore, besides introducing a brand new timer subsystem that copes with the scaling requirements of TCP support. This section highlights some notable technical details that characterize ClickNF implementation.

5.1 Packet I/O

Similarly to Fastclick [9], ClickNF provides fast packet I/O by using DPDK [33] to directly interface with network cards from user space. For packet reception, the DPDK element continuously poll the NIC to fetch received packet batches. In order to amortize the PCIe overhead, the DPDK element waits for a batch of 64 packets before transmitting them. To avoid head-of-line blocking and reduce latency, batches are transmitted after at most 100 μs. When needed, ClickNF performs batch processing (*i.e.,* Click elements process packet batches – implemented through packets' linked lists – instead of single packets) to optimize CPU cache performance. Also in this case, a batch is forwarded downstream after 100 μs even if it is not complete.

As in Fastclick, we modify the Click packet data structure to be a wrapper around a DPDK memory buffer (`mbuf`) to avoid additional memory allocation and copy operations. Each packet has a fixed size of 8 KB and consists of four sections, namely, the `mbuf` structure itself, packet annotations, headroom, and data. DPDK uses the `mbuf` for packet I/O whereas ClickNF uses annotations to store packet metadata (e.g., header pointers) and the headroom space to allow elements to prepend headers. Applications allocate a packet by filling only the data portion before pushing the packet down to lower layers.

Notice that, differently form Fastclick, we use a single element for both input and output operations in order to simplify the configuration. Moreover, ClickNF can also leverage common NIC features to perform flow control, TCP/IP checksum offloading, TCP segmentation (TSO), and large receive offloading (LRO) using hardware acceleration. Flow control prevents buffer overflows by slowing down transmitters when the RX buffer in the NIC becomes full. TCP/IP checksum offloading allows the NIC to compute header checksums in hardware. TSO segments a large packet into MTU-sized packets, whereas LRO aggregates multiple received packets into a large TCP segment before dispatching it to higher layers. All of these features can be toggled in ClickNF at run-time.

5.2 Multicore Scalability

Multithreading is implemented to exploit the processing power in multicore CPUs and improve scalability. We design per-core lock-free data structures aiming for high performance. Each core maintains dedicated packet pools, timers, transport, and application layer data structures. Receive Side Scaling (RSS) is used to distribute packets to different cores according to their flow identifier, *i.e.,* flow 5-tuple. Each DPDK thread is pinned to a CPU core and provided with a TX and a RX hardware queue at the NIC, preserving flow-level core affinity.

To avoid low-level CPU synchronization primitives each core maintains separate cache-aligned data structures. In case of multi-connection dependency, such as a proxy server establishing a connection on behalf of a client, the source port of the new outgoing connection is selected such that RSS maps it to the same core of the original connection, thus avoiding locks at the application level. Finally, each application-layer network function is spawned on multiple cores so that flows directed can be handled entirely on a specific core.

5.3 Timer Subsystem

Click implements its timer subsystem using a priority queue in which the root node stores the timer closer to expire. Given that TCP timers are often canceled before expiration, we implement a timing wheel scheduler for efficiency [47]. Its key advantage is that timing wheels schedule and cancel timers in $O(1)$, as opposed to $O(\log n)$ in priority queues currently used in Click.

A timing wheel is composed of n buckets, an index b, a timestamp t, and a tick granularity g (e.g., 1 ms). The timestamp t keeps the current time and the index b, points to its corresponding bucket. Each bucket contains timers expiring in the future, such that bucket b contains timers expiring within $[t, t+g)$, and so on. To schedule a timer, we must first find its corresponding bucket. For an expiration time e in the interval $t \leq e < t + ng$, its bucket index is computed as $\lfloor(e-t)/g\rfloor$. Each bucket contains a doubly linked list to store the timers expiring within the same interval. Therefore, once the index is computed, the timer is inserted at the end of the list of the bucket. To cancel a timer, the timer is just removed from the doubly linked list. Both operations are done in $O(1)$ with simple modulo operation to compute the bucket index b.

6 Evaluation

In this section, we evaluate ClickNF with three goals: (i) evaluate its performance through a series of microbenchmarks; (ii) compare it against Linux and state-of-the-art user space stacks; and (iii) showcase the usage of ClickNF and its performance when building network functions. Our evaluation setup consists of 3 machines with Intel Xeon® 40-core E5-2660 v3 2.60GHz processors, 64 GB RAM, each equipped with an Intel® 82599ES network card containing two 10 GbE interfaces. The machines run Ubuntu 16.10 (GNU/Linux 4.4.0-51-generic x86_64), Click 2.1, and DPDK 17.02.

6.1 Microbenchmarks

We start by analyzing individual aspects of our system using microbenchmarks, including packet I/O throughput, the cost of modularity, and the impact of hardware offloading. We then evaluate two applications, namely bulk transfer and echo server, to understand the system performance in common scenarios. Unless otherwise specified, the experiments presented in this section are performed on single-core ClickNF instances.

Packet I/O and the cost of modularity: To evaluate the throughput of our DPDK element, we run a set of tests with the DPDK traffic generator (DPDK-TG) [25] on one side and ClickNF on the other one. For ClickNF, we use four configurations that respectively generate and immediately discard (no I/O), receive (RX), forward (FW), and transmit (TX) 64-byte packets. Finally, we evaluate the cost of modularity by adding *PushNull* elements that receive packets on the input port and send them on the output port without doing anything else.

Figure 7 presents the average throughput for the different scenarios with a series of *PushNull* elements. Without I/O, ClickNF throughput is limited by the CPU at 43 Mpps. Increasing the number of elements increases the time spent by a packet inside the Click graph, considerably reducing the system throughput. For the RX, FW, and TX scenarios, the throughput is limited by the NIC line rate at 14.88 Mpps. ClickNF is still able to sustain the line rate with up to 15–20 *PushNull* elements. At this point, the throughput is limited by CPU and decreases further as more elements are placed in the configuration.

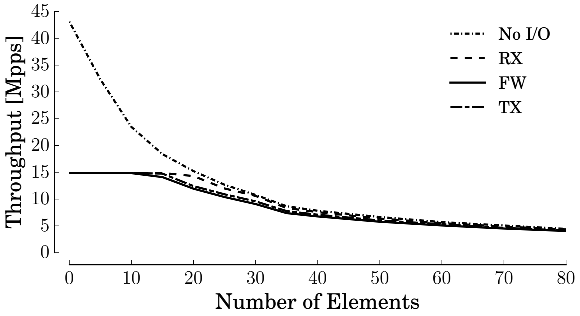


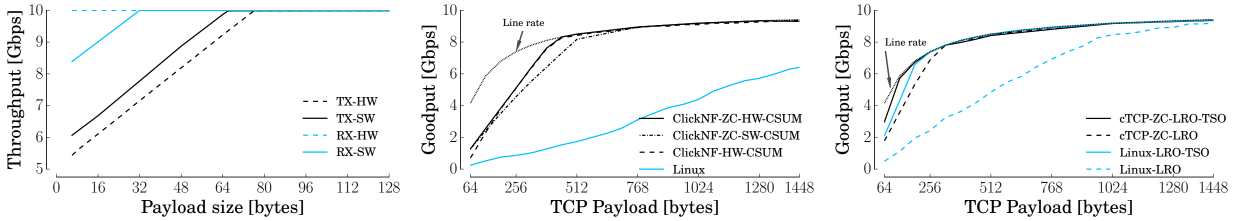
Figure 7: Throughput with 64-byte packets and increasing number of Click elements in different scenarios.

However, using packet batches between Click elements (Section 5.1) reduces the cost of modularity by improving instruction and data cache utilization. Indeed, when processing batches instead of single packets we experimentally evaluate that ClickNF to sustain line rate with up to 150 *PushNull* elements in RX, TX, and FW configurations. In ClickNF we adopt batch processing with batches of 32 packets to improve performance.

Checksum offloading: To evaluate hardware checksum offloading in our DPDK module, we measure the throughput using software or hardware checksum computation-verification. As in the previous tests, we run the DPDK-TG on one side and ClickNF on the other. We then use two different configurations for transmitting (TX) and receiving (RX) packets belonging to a single TCP flow. For transmissions, ClickNF computes header checksums before transmitting the packets to the traffic generator. For receptions, ClickNF verifies whether the checksums are correct before discarding the packet. Both operations are performed in hardware or in software.

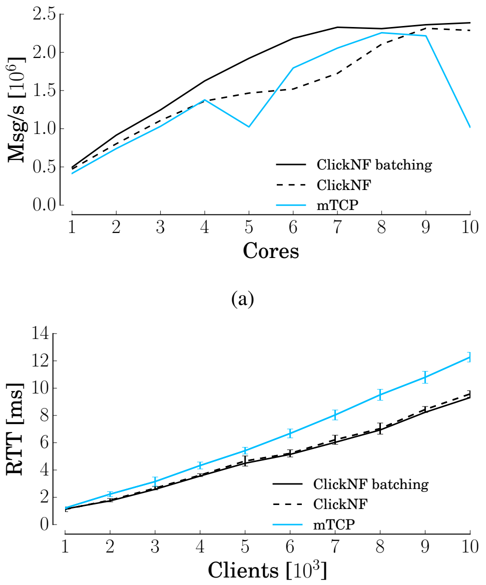
Figure 8a shows the results for TX and RX with increasing payload size (6–128 bytes). As expected, offloading checksum verification provides significant performance benefits and allows ClickNF to sustain line rate even when receiving small packets. Surprisingly, TX checksum computation in software is significantly better than in hardware. This is because modern CPUs are very efficient when performing sequential operations on cached memory, and sometimes even faster than dedicated hardware. However, as we see in the next experiments, this only holds because the CPU is underloaded.

Bulk transfer: To validate ClickNF in a more realistic scenario we execute a bulk transfer of a 20 GB file from a client to a server. Moreover, we evaluate the performance of TCP segment offloading (TSO) and large received offloading (LRO), as well as equivalent implementations in software Click elements. For comparison, we use iperf [4] running on top of Linux network.



(a) (b) (c)

Figure 8: (a) TCP throughput with and without TCP/IP checksum offloading. (b) TCP goodput in a bulk transfer with increasing TCP payload size. (c) TCP goodput in a bulk transfer with TSO and LRO enabled.



(a)

(b)

Figure 9: Comparison between ClickNF and mTCP. (a) Echo server message rate with increasing number of cores. (b) RTT with increasing number of TCP clients.

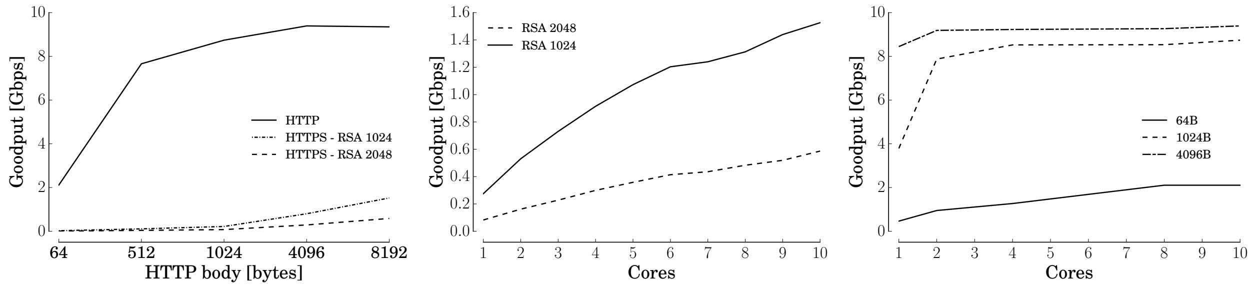
Figure 8b shows the TCP goodput for increasing payload sizes for ClickNF and Linux. For ClickNF, we use the socket API with HW checksum computation-verification (ClickNF-HW-CSUM), the zero-copy API with either software checksum computation-verification (ClickNF-ZC-SW-CSUM) or hardware checksum offloading (ClickNF-ZC-HW-CSUM). For Linux, we use iperf for tests with hardware checksum enabled. ClickNF significantly outperforms Linux and achieves line rate for TCP payloads larger than 448 bytes when the packet header overhead is smaller. For 64-byte payload, the zero-copy API provides roughly 50% throughput improvement over the socket API. This occurs because, with larger packets, the number of calls to memcopy and recv decreases, limiting the advantage of zero copy. In the following tests, we use the zero-copy API, as it delivers better performance with respect to the socket API.

Unlike what is observed in reception, TX checksum offloading (ClickNF-ZC-HW-CSUM) provides significant benefits with respect to software (ClickNF-ZC-SW-CSUM). For computationally intensive workloads, TX checksum offloading prevents the CPU from spending precious cycles in checksum computation-verification.

Figure 8c presents the results with TSO and LRO enabled. ClickNF outperforms Linux, specially for small payloads suggesting that the Linux stack is particularly inefficient for small packets, as reported in [20, 35, 10]. ClickNF achieves line rate for TCP payloads larger than 128 bytes while Linux have similar performance with TSO and LRO for TCP payloads larger than 192 bytes. In the following, we always enable LRO and TSO to amortize segmentation and reassembly cost.

Echo server: We also evaluate ClickNF in the presence of short TCP connections and compare its performance against mTCP [20], a user-space network stack with particular focus on performance and with similar goals to ClickNF. To evaluate multicore scalability, we run an echo server on top of both stacks. Clients running in two separate 8-core ClickNF instances connect to the server, send a 64-byte message, and wait for the echo reply. When the client receives the message back, it resets the connection to avoid port exhausting. The client then repeats the operation and measures the message rate. To provide a fair comparison against mTCP, we disable delayed ACKs that, when enabled, decrease the overhead and increase the overall throughput. Figure 6 depicts the configuration graph of the echo server used in this test, except for the *SSLServer* element that is not included.

First we measure the rate obtained by ClickNF and mTCP with a single core. ClickNF provides high throughput, 0.5×10^6 Msg/s, when using DPDK packet I/O. Compared to legacy Click elements for packet I/O, 0.169×10^6 Msg/s, (*i.e.,* using PCAP library linked to *FromDevice*), ClickNF provides 4x higher throughput. This shows how important kernel bypass is when enabling zero-copy packet processing at user space. Additionally, ClickNF outperforms mTCP, 0.415×10^6 Msg/s, by approximately 20%.



(a) HTTP cache - 10 cores with/without SSL/TLS (b) HTTP cache - with SSL/TLS (c) Proxy

Figure 10: Average goodput of the ClickNF modular HTTP(S) cache server.

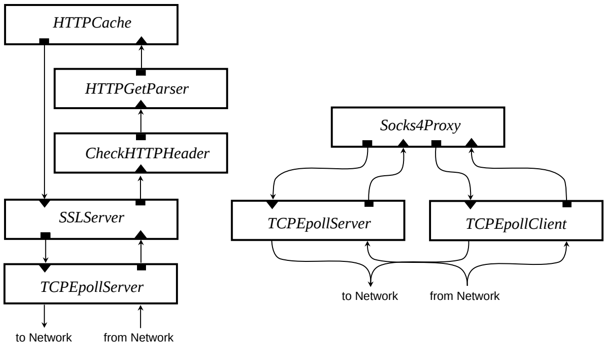


Figure 11: Configuration graphs for an HTTP cache server with SSL/TLS and for a SOCKS4 Proxy.

Figure 9a presents the message rate obtained with ClickNF and with mTCP for increasing number of cores. Despite its modularity, ClickNF provides equivalent performance to mTCP up to 4 cores, and scales slower for more cores. As observed earlier, modularity has a cost, but can be amortized with packet batches. In this case, Click elements receives packet batches instead of single packets hence optimizing CPU instruction and data cache usage. We enable batching in ClickNF for L2–L3 operations to avoid packet reordering issues at L4 and repeated the echo server experiment. Figure 9a shows that, ClickNF outperforms mTCP and achieves line rate with 7 cores. Results with hyperthreading (not reported here) show higher throughput, reducing the number of cores required to saturate the link to 4.

Since ClickNF employs batching at all levels, the risk is that RTT might be undesirably long. Figure 9b shows the RTT experienced by ClickNF and mTCP in the single core echo server test with increasing number of concurring clients. Due to the usage of batch timeouts, the latency introduced with increasing number of clients is limited and lower when compared to mTCP.

6.2 Modular Network Functions

To evaluate ClickNF performance and show the benefits of its modularity, we build two sample applications.

HTTP(S) cache: Figure 11 depicts the configuration graph used to deploy an HTTP cache server. Using the basic building blocks provided by ClickNF, application logic is implemented with three simple elements.

To evaluate the performance of our modular HTTP cache server, we run tests with SSL/TLS termination using self-signed certificates for 1024- or 2048-byte RSA keys. Clients running in two 8-core ClickNF instances first connect to the server and then issue HTTP GET requests for web pages of size 64–8192 bytes, stored in the HTTP cache server's main memory. The server responds to the requests and then closes the TCP connection.

Figure 10a,10b presents the goodput of the ClickNF HTTP cache server with and without SSL/TLS termination. With unencrypted HTTP traffic running on 10 cores (cf. Figure 10a), ClickNF achieves high goodput for small HTTP pages, and scales linearly with bigger page size. With SSL/TLS termination, the goodput drops to a maximum of $\approx$ 1.6 Gbps for 1024-byte keys and 10 cores. This is due to the complexity of public-key RSA cryptographic operations during SSL/TLS handshake [15]. This overhead, however, can be alleviated by delegating such operations to GPUs [27, 48].

SOCKS4 proxy: Socket Secure (SOCKS) is a protocol for enabling client-server communication through a proxy. Starting from a basic SOCKS4 proxy implementation written in C, we built a modular SOCKS4 proxy composed by three elements namely *Socks4Proxy*, and ClickNF building blocks *TCPEpollServer* and *TCPEpollClient*. Figure 11 depicts the high-level configuration graph for the proxy. Notice that, due to ClickNF L7 modularity, the SOCKS4 proxy graph can be easily modified to introduce additional functions (e.g., firewall, SSL encryption) right before the paths connecting *TCPEpollServer* and *Socks4Proxy*.

To evaluate the performance of our modular SOCKS4 proxy, we run a simple test similar to the one presented for the HTTP cache. In this case, clients connect to the SOCKS4 proxy which opens a new connection toward the HTTP cache. Once the connection is established, the client requests an HTTP page of fixed size and then resets the connection. Figure 10c presents the goodput of the ClickNF SOCKS4 proxy with increasing number of cores and variable HTTP body sizes. Similarly to the HTTP cache experiment, when the HTTP message is small the overhead (connection establishment and SOCKS protocol) is significant and prevents the system from achieving a higher goodput. For larger page sizes, the overhead decreases and the proxy is able to achieve close to line rate using just two CPU cores.

7 Related Work

Click [29] and its modular data plane have been improved and extended in multiple directions over almost two decades. For instance, Routebricks [18] parallelizes routing functionality across and within software routers building on top of Click to scale its performance. Recently, Fastclick [9] introduced high-speed packet I/O such as DPDK and netmap [33, 44] in Click.Moreover, GPU offloading is also proposed in [28, 46] to increase throughput beyond CPU capabilities. Similarly, ClickNP [32] provides Click-like programming abstractions to enable the construction of FPGA-accelerated network functions. ClickNF is orthogonal to such extensions and enables the modular composition of L2–L7 stacks, bridging Click's L2-L3 packet processing with L4 flow processing and L7 modular applications.

Click inspired other modular network function frameworks [8, 12, 40]. These systems mainly focus on control plane operations, such as data plane element placement, network function scaling, and traffic steering. For the data plane, FlowOS [11] is proposed as a middlebox platform that enables flow processing, but without TCP support. CoMb [45] and xOMB [7] use Click to consolidate middleboxes through the composition of different L7 elements. Both rely on the OS for packet I/O and transport layer, reducing customization and performance. Frameworks to enable stack customization of L2–L7 are proposed in [42, 26]. In [42], authors introduce an overview of the control and data planes of a modular architecture, with focus on hardware acceleration. In [26], the design of a modular middlebox platform based on mTCP [20] is presented. Instead of redesigning a framework with Click-like abstractions and/or providing new event-driven domain-specific APIs, ClickNF introduces L2–L7 modularity in Click to expose and exploit its modularity, as well as benefiting from existing Click extensions and contributions by the community.

Network stacks were proposed to overcome the I/O inefficiencies of OS [10, 43, 20, 35, 23, 49, 16]. IX [10] separates the control plane and data plane processing. Arrakis [43] is a customized OS that provide applications with direct access to I/O devices, allowing kernel bypass for I/O operations. mTCP [20] is a user-level TCP implementation proposed for multicore systems. It provides a socket API for application development supporting L2–L4 zero copy. In a different spirit, Stackmap [49] provides packet I/O acceleration to TCP kernel implementation obtaining better performance compared to Linux TCP. Sandstorm [35] proposes a specialized network stack with zero-copy APIs merging application and network logics. Similarly to ClickNF, Modnet [41] has a modular approach for providing customizable networking stack, but modularity is limited to L2–L4. Few other efforts [3, 5, 2, 16, 13] also provide efficient, sometimes modular, networking stacks but cannot completely benefit of L2–L7 modularity provided by ClickNF.

Our previous workshop paper [31] focused on providing an initial architecture for a modular TCP implementation in Click. ClickNF extends it in several directions. For instance, ClickNF takes advantage of hardware offloading, multicore scalability, timing wheels, and an epoll-based API to improve performance. Application level modularity and SSL/TLS termination provide building blocks for novel network functions to be deployed with little effort. We hereby propose a more comprehensive picture of ClickNF's performance, flexibility, and ease of use.

8 Conclusions

The advent of NFV gives us a different perspective on the way application servers and middleboxes can be implemented. ClickNF enables the composition of high-performance network functions at all layers of the network stack and opens up its inner workings for the benefit of developers. In this paper, we illustrated and benchmarked several concrete examples where ClickNF can be used to accelerate network function development by enabling fine-grained code reuse, and highlighted ClickNF's good scaling properties and a reasonable price of modularity, which arguably outweigh many of the hurdles in network function development. The ClickNF source code is available for download at [21].

Acknowledgments

We would like to thank our anonymous reviewers that provided several advices to improve and further develop our work as well as Fabio Pianese for the fruitful discussions and for his countless paper revisions.

References

[1] The Linux Kernel. /urlhttps://www.kernel.org/.

[2] Open Fast path, 2015. `http://www.openfastpath.org/`.

[3] 6windgate, 2017. `http://www.6wind.com/products/6windgate/`.

[4] Iperf2, 2017. `https://sourceforge.net/projects/iperf2/`.

[5] The Fast Data Project FD.io, 2017. `https://fd.io/`.

[6] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference* (2010), SIGCOMM '10.

[7] Anderson, J. W., Braud, R., Kapoor, R., Porter, G., and Vahdat, A. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2012), ANCS '12, ACM.

[8] Anwer, B., Benson, T., Feamster, N., and Levin, D. Programming Slick Network Functions. In *Proc. 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (2015), SOSR '15.

[9] Barbette, T., Soldani, C., and Mathy, L. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2015), ANCS '15, IEEE Computer Society.

[10] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. 11th USENIX OSDI* (2014).

[11] Bezahaf, M., Alim, A., and Mathy, L. FlowOS: A Flow-based Platform for Middleboxes. In *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (2013), HotMiddlebox '13, ACM.

[12] Bremler-Barr, A., Harchol, Y., and Hay, D. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *Proc. of the 2016 ACM SIGCOMM Conference* (2016).

[13] Bridges, P. G., Wong, G. T., Hiltunen, M., Schlichting, R. D., and Barrick, M. J. A configurable and extensible transport protocol. *IEEE/ACM Transactions on Networking 15*, 6 (2007), 1254–1265.

[14] Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H., and Jacobson, V. Bbr: Congestion-based congestion control. *Queue 14*, 5 (2016), 50:20–50:53.

[15] Coarfa, C., Druschel, P., and Wallach, D. S. Performance analysis of TLS web servers. *ACM Transaction of Compututer System* (2006).

[16] Condie, T., Hellerstein, J. M., Maniatis, P., Rhea, S., and Roscoe, T. Finally, a use for componentized transport protocols. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2005), HotNets-IV.

[17] Cronkite-Ratcliff, B., Bergman, A., Vargaftik, S., Ravi, M., McKeown, N., Abraham, I., and Keslassy, I. Virtualized congestion control. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), SIGCOMM '16.

[18] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), SOSP '09, ACM.

[19] ETSI. Network Function Virtualization. *SDN & OpenFlow World Congress* (2014).

[20] EunYoung, J., Shinae, W., Muhammad, J., Haewon, J., Sunghwan, I., Dongsu, H., and KyoungSoo, P. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. 11th USENIX NSDI* (2014).

[21] Gallo, M., and Laufer, R. The ClickNF framework. Available on `https://github.com/nokia/ClickNF`, 2017.

[22] He, K., Rozner, E., Agarwal, K., Gu, Y. J., Felter, W., Carter, J., and Akella, A. Ac/dc tcp: Virtual congestion control enforcement for datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), SIGCOMM '16.

[23] Hruby, T., Giuffrida, C., Sambuc, L., Bos, H., and Tanenbaum, A. S. A NEaT Design for Reliable and Scalable Network Stacks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* (2016), CoNEXT '16, ACM.

[24] Hwang, J., Ramakrishnan, K. K., and Wood, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. *IEEE Transactions on Network and Service Management* (2015).

[25] Intel. DPDK Traffic Generator, 2017. `http://dpdk.org/browse/apps/pktgen-dpdk/`.

[26] Jamshed, M. A., Moon, Y., Kim, D., Han, D., and Park, K. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proc. of 14th USENIX NSDI* (2017).

[27] Jang, K., Han, S., Han, S., Moon, S., and Park, K. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proc. 8th USENIX NSDI* (2011).

[28] Kim, J., Jang, K., Lee, K., Ma, S., Shim, J., and Moon, S. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), EuroSys '15, ACM.

[29] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click Modular Router. *ACM Transaction of Computer System* (2000).

[30] Langley, A. e. a. The quic transport protocol: Design and internet-scale deployment. In *Proc. of the 2017 ACM SIGCOMM Conference* (2017).

[31] Laufer, R., Gallo, M., Perino, D., and Nandugudi, A. CliMB: Enabling network function composition with Click middleboxes. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (2016), HotMIddlebox '16, ACM.

[32] Li, B., Tan, K., Luo, L. L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proc. of the 2016 ACM SIGCOMM Conference* (2016).

[33] Linux Foundation. DPDK, 2017. http://dpdk.org.

[34] Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., and Huici, F. My vm is lighter (and safer) than your container. In *Proceedings of the Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 218–233.

[35] Marinos, I., Watson, R. N., and Handley, M. Network Stack Specialization for Performance. In *Proc. of the 2014 ACM SIGCOMM Conference* (2014), SIGCOMM '14, ACM.

[36] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. ClickOS and the Art of Network Function Virtualization. In *Proc. 11th USENIX NSDI* (2014).

[37] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* (2008).

[38] Narayan, A., Cangialosi, F., Goyal, P., Narayana, S., Alizadeh, M., and Balakrishnan, H. The case for moving congestion control out of the datapath. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), HotNets-XVI.

[39] Niu, Z., Xu, H., Han, D., Cheng, P., Xiong, Y., Chen, G., and Winstein, K. Network stack as a service in the cloud. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), HotNets-XVI.

[40] Palkar, S., Lan, C., Han, S., Jang, K., Panda, A., Ratnasamy, S., Rizzo, L., and Shenker, S. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, ACM.

[41] Pathak, S., and Pai, V. S. Modnet: A modular approach to network stack extension. In *Proc. 12th USENIX NSDI* (2015).

[42] Perino, D., Gallo, M., Laufer, R., Houidi, Z. B., and Pianese, F. A Programmable Data Plane for Heterogeneous NFV Platforms. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (2016).

[43] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The Operating System is the Control Plane. In *Proc. 11th USENIX OSDI* (2014).

[44] Rizzo, L. netmap: A Novel Framework for Fast Packet I/O. In *Proc. of the 2012 USENIX Annual Technical Conference* (2012).

[45] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of 9th USENIX NSDI* (2012).

[46] Sun, W., and Ricci, R. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2013), ANCS '13, IEEE Press.

[47] Varghese, G., and Lauck, A. Hashed and hierarchical timing wheels: Efficient data structures for implementing a timer facility. *IEEE/ACM Transactions on Networking* (1997).

[48] Varvello, M., Laufer, R., Zhang, F., and Lakshman, T. Multilayer packet classification with graphics processing units. *IEEE/ACM Transactions on Networking* (2016).

[49] Yasukata, K., Honda, M., Santry, D., and Eggert, L. Stackmap: Low-latency networking with the OS stack and dedicated nics. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016).

Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics

Ana Klimovic
Stanford University

Heiner Litz
UC Santa Cruz

Christos Kozyrakis
Stanford University

Abstract

Data analytics are an important class of data-intensive workloads on public cloud services. However, selecting the right compute and storage configuration for these applications is difficult as the space of available options is large and the interactions between options are complex. Moreover, the different data streams accessed by analytics workloads have distinct characteristics that may be better served by different types of storage devices.

We present Selecta, a tool that recommends near-optimal configurations of cloud compute and storage resources for data analytics workloads. Selecta uses latent factor collaborative filtering to predict how an application will perform across different configurations, based on sparse data collected by profiling training workloads. We evaluate Selecta with over one hundred Spark SQL and ML applications, showing that Selecta chooses a near-optimal performance configuration (within 10% of optimal) with 94% probability and a near-optimal cost configuration with 80% probability. We also use Selecta to draw significant insights about cloud storage systems, including the performance-cost efficiency of NVMe Flash devices, the need for cloud storage with support for fine-grain capacity and bandwidth allocation, and the motivation for end-to-end storage optimizations.

1 Introduction

The public cloud market is experiencing unprecedented growth, as companies move their workloads onto platforms such as Amazon AWS, Google Cloud Platform and Microsoft Azure. In addition to offering high elasticity, public clouds promise to reduce the total cost of ownership as resources can be shared among tenants. However, achieving performance and cost efficiency requires choosing a suitable configuration for each given application. Unfortunately, the large number of instance types and configuration options available make selecting the right resources for an application difficult.

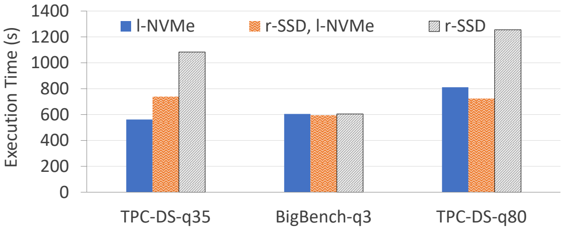


Figure 1: Performance of three applications on eight `i3.xl` instances with different storage configurations.

The choice of storage is often essential, particularly for cloud deployments of data-intensive analytics. Cloud vendors offer a wide variety of storage options including object, file and block storage. Block storage can consist of hard disks (*HDD*), solid-state drives (*SSD*), or high bandwidth, low-latency NVMe Flash devices (*NVMe*). The devices may be local (*l*) to the cloud instances running the application or remote (*r*). These options alone lead to storage configuration options that can differ by orders of magnitude in terms of throughput, latency, and cost per bit. The cloud storage landscape is only becoming more diverse as emerging technologies based on 3D X-point become available [35, 16].

Selecting the right cloud storage configuration is critical for both performance and cost. Consider the example of a Spark SQL equijoin query on two 128 GB tables [53]. We find the query takes 8.7× longer when instances in an 8-node EC2 cluster access *r*-HDD compared to *l*-NVMe storage. This is in contrast to a recent study, conducted with a prior version of Spark, which found that faster storage can only improve the median job execution time by at most 19% [50]. The performance benefits of *l*-NVMe lead to 8× lower execution cost for this query, even though NVMe storage has higher cost per unit time. If we also consider a few options for the number of cores and memory per instance, the performance gap between the best and worst performing VM-storage configurations is over 30×.

Determining the right cloud configuration for analytics applications is challenging. Even if we limit ourselves to a single instance type and focus on optimizing performance, the choice of storage configuration for a particular application remains non-trivial. Figure 1 compares the performance of three Spark applications using 8 `i3.xl` AWS instances with *l*-NVMe, *r*-SSD, and a hybrid (*r*-SSD for input/output data, *l*-NVMe for intermediate data). The first application is I/O-bound and benefits from the high throughput of NVMe Flash. The second application has a CPU bottleneck and thus performs the same with all three storage options. The third application is I/O-bound and performs best with the hybrid storage option since it minimizing interference between read and write I/Os, which have asymmetric performance on Flash [40]. This result should not be surprising. Analytics workloads access multiple data streams, including input and output files, logs, and intermediate data (e.g., shuffle and broadcast). Each data stream has distinct characteristics in terms of access frequency, access patterns, and data lifetime, which make different streams more suitable for different types of storage devices. For example, for TPC-DS query 80 in Figure 1, storing input/output data on *r*-SSD and intermediate data on *l*-NVMe Flash outperforms storing all data on *l*-NVMe as it isolates streams and eliminates interference.

We present *Selecta*, a tool that learns near-optimal VM and storage configurations for analytics applications for user-specified performance-cost objectives. Selecta targets analytics jobs that are frequently or periodically rerun on newly arriving data [1, 25, 55]. A configuration is defined by the type of cloud instance (core count and memory capacity) along with the storage type and capacity used for input/output data and for intermediate data. To predict application performance for different configurations, Selecta applies latent-factor collaborative filtering, a machine-learning technique commonly used in recommender systems [10, 57, 11, 22, 23]. Selecta uses sparse performance data for training applications profiled on various cloud configurations, as well as performance measurements for the target application profiled on only two configurations. Selecta leverages the sparse training data to learn significantly faster and more cost-effectively than exhaustive search. The approach also improves on recent systems such as CherryPick and Ernest whose performance prediction models require more information about the target application and hence require more application runs to converge [3, 69]. Moreover, past work does not consider the heterogeneous cloud storage options or the varying preferences of different data streams within each application [71].

We evaluate Selecta with over one hundred Spark SQL and ML workloads, each with two different dataset scaling factors. We show that Selecta chooses a near-optimal performance configuration (within 10% of optimal) with 94% probability and a near-optimal cost configuration with 80% probability. We also analyze Selecta's sensitivity to various parameters such as the amount of information available for training workloads or the target application.

A key contribution of our work is our analysis of cloud storage systems and their use by analytics workloads, which leads to several important insights. We find that in addition to offering the best performance, NVMe-based configurations also offer low execution cost for a wide range of applications. We observe the need for cloud storage options that support fine-grain allocation of capacity and bandwidth, similar to the fine-grain allocation of compute and memory resources offered by serverless cloud services [7]. Disaggregated NVMe Flash can provide the substrate for such a flexible option for cloud storage. Finally, we showcase the need for end-to-end optimization of cloud storage, including application frameworks, operating systems, and cloud services, as several storage configurations fail to meet their potential due to inefficiencies in the storage stack.

2 Motivation and Background

We discuss current approaches for selecting a cloud storage configuration and explain the challenges involved.

2.1 Current Approaches

Conventional configurations: Input/output files for data analytics jobs are traditionally stored in a distributed file system, such as HDFS or object storage systems such as Amazon S3 [62, 6]. Intermediate data is typically read/written to/from a dedicated local block storage volume on each node (i.e., *l*-SSD or *l*-NVMe) and spilled to *r*-HDD if extra capacity is needed. In typical Spark-as-a-service cloud deployments, two remote storage volumes are provisioned by default per instance: one for the instance root volume and one for logs [19].

Existing tools: Recent work focuses on automatically selecting an optimal VM configuration in the cloud [71, 69, 3]. However, these tools tend to ignore the heterogeneity of cloud storage options, at best distinguishing between 'fast' and 'slow'. In the next section, we discuss the extent of the storage configuration space.

2.2 Challenges

Complex configuration space: Cloud storage comes in multiple flavors: object storage (e.g., Amazon S3 [6]), file storage (e.g., Azure Files [45]), and block storage (e.g., Google Compute Engine Persistent Disks [29]). Block and object storage are most commonly used for data analytics. Block storage is further sub-divided into hardware options: cold or throughput-optimized hard drive disk, SAS SSD, or NVMe Flash. Block storage can be local (directly attached) or remote (over the network) to an instance. Local block storage is ephemeral; data persists only as long as the instance is running. Remote volumes persist until explicitly deleted by the user.

Table 1 compares three block storage options available in Amazon Web Services (AWS). Each storage option provides a different performance, cost, and flexibility trade-off. For instance, *l*-NVMe storage offers the highest throughput and lowest latency at higher cost per bit. Currently, cloud providers typically offer NVMe in fixed capacity units directly attached to select instance types, charged per second or hour. AWS currently charges $0.023 more per hour for an instance with 475 GB of NVMe Flash compared to without NVMe. In contrast, S3 fees are based on capacity ($0.023 per GB/month) and bandwidth ($0.004 per 10K GET requests) usage.

In addition to the storage configuration, users must choose from a variety of VM types to determine the right number of CPU cores and memory, the number of VMs, and their network bandwidth. These choices often affect storage and must be considered together. For example, on instances with 1 Gb/s network bandwidth, the network limits the sequential throughput achievable with *r*-HDD and *r*-SSD storage volumes in Table 1.

Performance-cost objectives: While configurations with the most CPU cores, the most memory, and fastest storage generally provide the highest performance, optimizing for runtime cost is much more difficult. Systems designed to optimize a specific objective (e.g., predict the configuration that maximizes performance or minimizes cost) are generally not sufficient to make recommendations for more complex objectives (e.g., predict the configuration that minimizes execution time within a specific budget). By predicting application execution time on candidate configurations, our approach remains general. Unless otherwise specified, we refer to cost as the cost of executing an application.

Heterogeneous application data: We classify data managed by distributed data analytics frameworks (e.g., Spark [74]) into two main categories: *input/output data* which is typically stored long-term and *intermediate data* which lives for the duration of job execution. Examples of intermediate data include shuffle data exchanged between mappers and reducers, broadcast variables, and cached dataset partitions spilled from memory. These streams typically have distinct access frequency, data lifetime, access type (random vs. sequential), and I/O size. For example, input/output data is generally long-lived and sequentially accessed, whereas intermediate data is short-lived and most accesses are random.

Storage	Seq Read MB/s	Seq Write MB/s	Rand Read IOPS	Rand Write IOPS	Rand Rd/Wr IOPS
r-HDD	135	135	132	132	132
r-SSD	165	165	3,068	3,068	3,068
l-NVMe	490	196	103,400	35,175	70,088

Table 1: Block storage performance for 500GB volumes. Sequential IOs are 128 KB, random IOs are 4 KB.

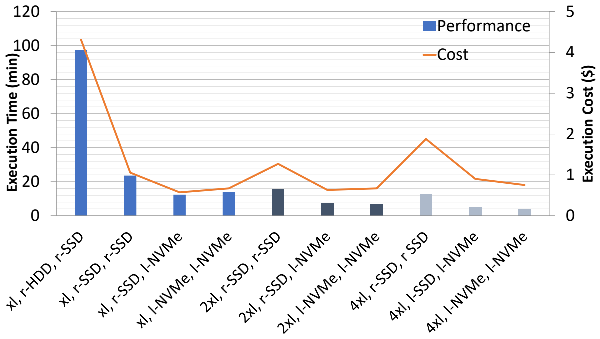


Figure 2: Comparison of execution time and cost for TPC-DS query 64 on various VM and storage configurations, defined as <VM size, storage for input/output data, storage for intermediate data>.

Storage decisions are complex: Selecting the right configuration for a job significantly reduces execution time and cost, as shown in Figure 2, which compares a Spark SQL query (TPC-DS query 64) on various VM and storage configurations in an 8-node cluster. We consider 3 `i3` VM instance sizes in EC2 (`xl, 2xl, and 4xl`) and heterogeneous storage options for input/output and intermediate data. The lowest performing configuration has 24× the execution time of the best performing configuration. Storing input/output data on *r*-SSD and intermediate data on *l*-NVMe (the lowest cost configuration) has 7.5× lower cost than storing input/output data on *r*-HDD and intermediate data on *r*-SSD.

3 Selecta Design

3.1 Overview

Selecta is a tool that automatically predicts the performance of a target application on a set of candidate configurations. As shown in Figure 3, Selecta takes as input: i) execution time for a set of training applications on several configurations, ii) execution time for the target application on two reference configurations, and iii) a performance-cost objective for the target application. A configuration is defined by the number of nodes (VM instances), the CPU cores and memory per node, as well as the storage type and capacity used for input/output data and for intermediate data. Selecta uses latent factor collaborative filtering (see §3.2) to predict the performance

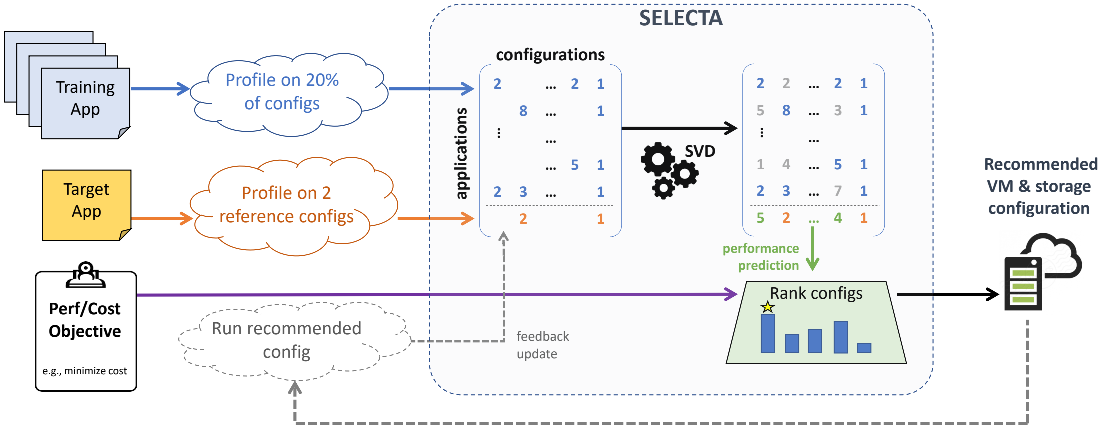


Figure 3: An overview of performance prediction and configuration recommendation with Selecta.

of the target application on the remaining (non-reference) candidate configurations. With these performance predictions and the per unit time cost of various VM instances and storage options, Selecta can recommend the right configuration for the user's performance-cost objective. For example, Selecta can recommend configurations that minimize execution time, minimize cost, or minimize execution time within a specific budget.

As new applications are launched over time, these performance measurements become part of Selecta's growing training set and accuracy improves (see § 4.4). We also feed back performance measurements after running a target application on a configuration recommended by Selecta — this helps reduce measurement noise and improve accuracy. Since Selecta takes ∼1 minute to generate a new set of predictions (the exact runtime depends on the training matrix size), a user can re-run Select when re-launching the target application with a new dataset to get a more accurate recommendation. In our experiments, the recommendations for each target application converge after two feedback iterations. The ability to grow the training set over time also provides Selecta with a mechanism for expanding the set of configurations it considers. Initially, the configuration space evaluated by Selecta is the set of configurations that appear in the original training set. When a new configuration becomes available and Selecta receives profiling data for applications on this configuration, the tool will start predicting performance for all applications on this configuration.

3.2 Predicting Performance

Prediction approach: Selecta uses collaborative filtering to predict the performance of a target application on candidate configurations. We choose collaborative filtering as it is agnostic to the details of the data analytics framework used (e.g., Spark vs. Storm) and it allows us to leverage *sparse* training data collected *across applications* and configurations [56]. While systems such as CherryPick [3] and Ernest [69] build performance models based solely on training data for the target application, Selecta's goal is to leverage training data available from multiple applications to converge to accurate recommendations with only two profiling runs of a target application. We discuss alternatives to collaborative filtering to explain our choice.

Content-based approaches, such as as linear regression, random forests, and neural network models, build a model from features such as application characteristics (e.g., GB of shuffle data read/written) and configuration characteristics (e.g., I/O bandwidth or the number of cores per VM). We find that unless inputs features such as the average CPU utilization of the target application *on the target configuration* are used in the model, content-based predictors do not have enough information to learn the compute and I/O requirements of applications and achieve low accuracy. Approaches that require running target applications on all candidate configurations to collect feature data are impractical.

Another alternative is to build performance prediction models based on the structure of an analytics framework, such as the specifics of the map, shuffle, and reduce stages in Spark [36, 75]. This leads to framework-specific models and may require re-tuning or even remodeling as framework implementations evolve (e.g., as the CPU efficiency of serialization operations improves).

Latent factor collaborative filtering: Selecta's collaborative filtering model transforms applications and configurations to a latent factor space [10]. This space characterizes applications and configurations in terms of latent (i.e., 'hidden') features. These features are *automatically inferred* from performance measurements of training applications [56]. We use a matrix factorization technique known as Singular Value Decomposition (SVD) for the latent factor model. SVD decomposes an input matrix P, with rows representing applications and columns representing configurations, into the product of three matrices, U, λ, and V. Each element p_{ij} of P represents the normalized performance of application i on configuration j. The latent features are represented by singular values in the diagonal matrix λ, ordered by decreasing magnitude. The matrix U captures the strength of the correlation between a row in P and a latent feature in λ. The matrix V captures the strength of the correlation between a column in P and a latent feature in λ. Although the model does not tell us what the latent features physically represent, a hypothetical example of a latent feature is random I/O throughput. For instance, Selecta could infer how strongly an application's performance depends on random I/O throughput and how much random I/O throughput a configuration provides.

One challenge for running SVD is the input matrix P is sparse, since we only have the performance measurements of applications on certain configurations. In particular, we only have two entries in the target application row and filling in the missing entries corresponds to predicting performance on the other candidate configurations. Since performing SVD matrix factorization requires a fully populated input matrix P, we start by randomly initializing the missing entries and then run Stochastic Gradient Descent (SGD) to update these unknown entries using an objective function that minimizes the mean squared error on the *known* entries of the matrix [13]. The intuition is that by iteratively decomposing and updating the matrix in a way that minimizes the error for known entries, the technique also updates unknown entries with accurate predictions. Selecta uses the Python sci-kit `surprise` library for SVD [33].

3.3 Using Selecta

New target application: The first time an application is presented to Selecta, it is profiled on two reference configurations which, preferably, are far apart in their compute and storage resource attributes. Selecta requires that reference configurations remain fixed across all applications, since performance measurements are normalized to a reference configuration before running SVD. Profiling application performance involves running the application to completion and recording execution time and CPU utilization (including iowait) over time.

Defining performance-cost objectives: After predicting application performance across all configurations, Selecta recommends a configuration based on a user-defined ranking function. For instance, to minimize runtime cost, the ranking function is min(runtime × cost/hour). While choosing a storage technology (e.g., SSD vs. NVMe Flash), Selecta must also consider the application's storage capacity requirements. Selecta leverages statistics from profiling runs available in Spark monitoring logs to determine the intermediate (shuffle) data and and input/output data capacity [63].

Adapting to changes: Recurring jobs and their input datasets are likely to evolve. To detect changes in application characteristics that may impact the choice of optimal configuration, Selecta relies on CPU utilization information from both initial application profiling and subsequent executions rounds. When an application is first introduced to the system, Selecta assigns a unique ID to store application specific information such as iowait CPU utilization. Whenever an application is re-executed, Selecta compares the current iowait time to the stored configuration. Depending on the difference in iowait time, Selecta will either compute a refined prediction based on available measurements or treat the workload as new application, starting a new profiling run.

Dealing with noise in the cloud: An additional challenge for recommending optimal configurations is noise on public cloud platforms, which arises due to interference with other tenants, hardware heterogeneity, or other sources [59]. To account for noise, Selecta relies on the feedback of performance and CPU utilization measurements. Initially, with few profiling runs, Selecta's performance predictions are affected by noise. As more measurements are fed into the system, Selecta averages performance and CPU utilization and uses reservoir sampling to avoid high skew from outliers [70]. Selecta keeps a configurable number of sample points for each entry in the application-configuration matrix (e.g., three) to detect changes in applications as described above. If a particular run is heavily impacted by noise such that the compute and I/O bottlenecks differ significantly from previous runs, Selecta's mechanism for detecting changes in applications identifies the outlier.

4 Selecta Evaluation

Selecta's collaborative filtering approach is agnostic to the choice of applications and configurations. We evaluate Selecta for data analytics workloads on a subset of the cloud configuration space with the goal of understanding how to provision cloud storage for data analytics.

4.1 Methodology

Cloud configurations: We deploy Selecta on Amazon EC2 and consider configurations with the instance and storage options shown in Tables 2 and 3. Among the possible VM and storage combinations, we consider seventeen candidate configurations. We trim the space to stay within our research budget and to focus on experiments that are most likely to uncover interesting insights about cloud storage for analytics. We choose EC2 instance families that are also supported by Databricks, a popular Spark-as-a-service provider [18]. `i3` is currently the only instance family available with NVMe Flash and

Instance	CPU cores	RAM (GB)	NVMe
`i3.xlarge`	4	30	1 x 950 GB
`r4.xlarge`	4	30	-
`i3.2xlarge`	8	60	1 x 1.9 TB
`r4.2xlarge`	8	60	-
`i3.4xlarge`	16	120	2 x 1.9 TB
`r4.4xlarge`	16	120	-

Table 2: AWS instance properties

Storage	Type	Locality	Use for Input/Output Data?	Use for Intermediate Data?
r-HDD	Block	Remote	✓	-
r-SSD	Block	Remote	✓	✓
l-NVMe	Block	Local	✓	✓
S3	Object	Remote	✓	-

Table 3: AWS storage options considered

`r4` instances allow for a fair comparison of storage options as they have the same memory to compute ratio. We only consider configurations where the intermediate data storage IOPS are equal to or greater than the input/output storage IOPS, as intermediate data has more random accesses. Since we find that most applications are I/O-bound with *r*-HDD, we only consider *r*-HDD for the instance size with the least amount of cores. We limit our analysis to *r*-HDD because our application datasets are up to 1 TB whereas instances with *l*-HDD on AWS come with a minimum of 6 TB disk storage, which would not be an efficient use of capacity. We do not consider local SAS/SATA SSDs as their storage capacity to CPU cores ratio is too low for most Spark workloads. We use Elastic Block Store (EBS) for remote block storage [5].

We use a cluster of 9 nodes for our evaluation. The cluster consists of one master node and eight executor nodes. The master node runs the Spark driver and YARN Resource Manager. Unless input/output data is stored in S3, we run a HDFS namenode on the master server as well. We configure framework parameters, such as the JVM heap size and number of executors, according to Spark tuning guidelines and match the number of executor tasks to the VM's CPU cores [15, 14].

Applications: We consider Spark [74] as a representative data analytics framework, similar to previous studies [50, 68, 3]. We use Spark v2.1.0 and Hadoop v2.7.3 for HDFS. We evaluate Selecta with over one hundred Spark SQL and ML applications, each with two different dataset scales, for a total of 204 workloads. Our application set includes 92 queries of the TPC-DS benchmark with scale factors of 300 and 1000 GB [67]. We use the same scale factors for Spark SQL and ML queries from the TPC-BB (BigBench) benchmark which has of structured, unstructured and semi-structured data modeled after the retail industry domain [27]. Since most BigBench queries are CPU-bound, we focus on eight queries which have more substantial I/O requirements: queries 3, 8, 14, 16, 21, 26, 28, 29. We also run 100 and 400 GB sort jobs [52]. Finally, we run a SQL equijoin query on two tables with 16M and 32M rows each and 4KB entries [53]. For all input and output files, we use the uncompressed Parquet data format [26].

Experiment methodology: We run each application on all candidate configurations to obtain the ground truth performance and optimal configuration choices for each application. To account for noise in the cloud we run each experiment (i.e., each application on each candidate configuration) three times and use the average across runs in our evaluation. Two runs are consecutive and one run is during a different time of day. We also validate our results by using data from one run as input to Selecta and the average performance across runs as the ground truth. To train and test Selecta, we use leave-one-out cross validation [58], meaning one workload at a time serves as the target application while the remaining workloads are used for training. We assume training applications are profiled on all candidate configurations, except for the sensitivity analysis in §4.4 where we investigate training matrix density requirements for accurate predictions.

Metrics: We measure the quality of Selecta's predictions using two metrics. First, we report the relative root mean squared error (RMSE), a common metric for recommender systems. The second and more relevant metric for Selecta is the probability of making an accurate configuration recommendation. We consider a recommendation accurate if the configuration meets the user's cost-performance objective within a threshold T of the true optimal configuration for that application. For example, for a minimum cost objective with T = 10%, the probability of an accurate prediction is the percentage of Selecta's recommendations (across all tested applications) whose true cost is within 10% of the true optimal cost configuration. Using a threshold is more robust to noise and allows us to make more meaningful conclusions about Selecta's accuracy, since a second-best configuration may have similar or significantly worse performance than the best configuration. Our performance metric is execution time and cost is in US dollars.

4.2 Prediction Accuracy

We provide a matrix with 204 rows as input to Selecta, where one row (application) is designated as the target application in each test round. We run Selecta 204 times, each time considering a different application as the target. For now, we assume all remaining rows of training data in the matrix are dense, implying the user has profiled training applications on all candidate configurations. The single target application row is sparse, containing only two entries, one for each of the profiling runs on reference configurations.

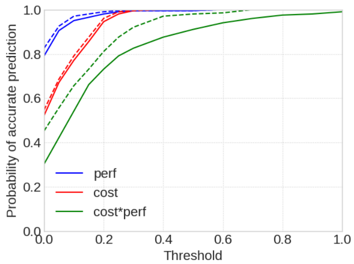


Figure 4: Probability of accurate recommendations within a threshold from optimal. Dotted lines are after one feedback iteration.

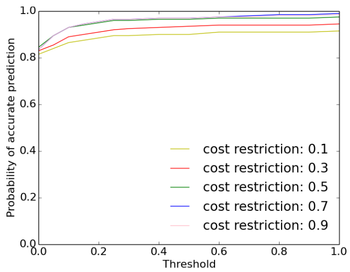


Figure 5: Probability of accurate configuration recommendation for performance within threshold, given strict cost restrictions.

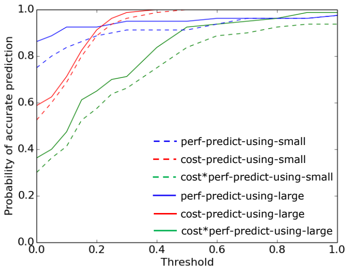


Figure 6: Accuracy with large datasets using predictions from small dataset vs. re-computing prediction with large dataset.

Selecta predicts performance with a relative RMSE of 36%, on average across applications. To understand how Selecta's performance predictions translate into recommendations, we plot accuracy in Figure 4 for performance, cost and cost*performance objectives. The plot shows the probability of near-optimal recommendations as a function of the threshold T defining what percentage from optimal is considered close enough. When searching for the best performing configuration, Selecta has a 94% probability of recommending a configuration within 10% of optimal. For a minimum cost objective, Selecta has a 80% probability of recommending a configuration within 10% of optimal. Predicting cost*performance is more challenging since errors in Selecta's relative execution time predictions for an application across candidate configurations are squared: cost*performance = (execution_time)^2 * config_cost_per_hour.

The dotted lines in Figure 4 show how accuracy improves after a single feedback round. Here, we assume the target application has the same dataset in the feedback round. This provides additional training input for the target application row (either a new entry if the recommended configuration was not a reference configuration, or a new sample to average to existing data if the recommended configuration was a reference configuration). The probability of near-optimal recommendations increases most noticeably for the cost*performance objective, from 52% to 65% after feedback, with T=10%.

Figure 5 shows the probability of accurate recommendations for objectives of the form "select the best performing configuration given a fixed cost restriction C." For this objective, we consider Selecta's recommendation accurate if its cost is less than or equal to the budget and if its performance is within the threshold of the true best configuration for the objective. Selecta achieves between 83% and 94% accuracy for the cost restrictions in Figure 5 assuming T=10%. The long tail is due to performance prediction errors that lead Selecta to underestimate the execution cost for a small percentage of configurations (i.e., cases where Selecta recommends a configuration that is actually over budget).

In Figure 7, we compare Selecta's accuracy against four baselines. The first baseline is a random forest predictor, similar to the approach used by PARIS [71]. We use the following features: the number of CPU cores, disk IOPS and disk MB/s the configuration provides, the intermediate and input/output data capacity of the application, and the CPU utilization, performance, and total disk throughput measured when running the application on each of the two reference configurations. Although the random forest predictor leverages more features than Selecta, it has lower accuracy. Collaborative filtering is a better fit for the sparse nature of the training data. We find the most important features in the random forest model are all related to I/O (e.g., the I/O throughput measured when running the application on the reference configurations and the read/write IOPS supported by the storage used for intermediate data), which emphasizes the importance of selecting the right storage.

The second baseline (labeled 'default') in Figure 7 uses the recommended default configurations documented in Databricks engineering blog posts: *l*-NVMe for intermediate data and S3 for input/output data [19, 21, 20]. The 'max cost per time' baseline uses the simple heuristic of always picking the most expensive instance per unit time. The 'min cost per time' baseline chooses the least expensive instance per unit time. Selecta outperforms all of these heuristic strategies, confirming the need for a tool to automate configuration selection.

4.3 Evolving Datasets

We study the impact of dataset size on application performance and Selecta's predictions using the small and large dataset scales described in §4.1. We train Selecta using all 102 workloads with small datasets, then evaluate Selecta's prediction accuracy for the same workloads with large datasets. The dotted lines in Figure 6 plots Se-

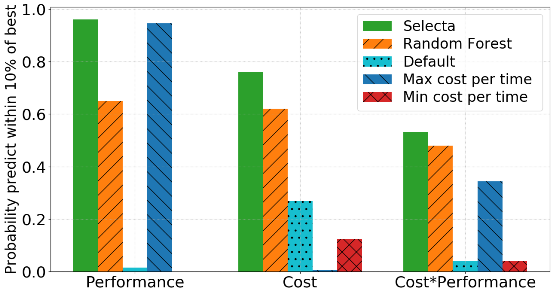
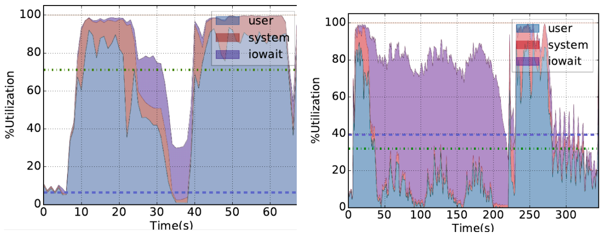


Figure 7: Selecta's accuracy compared to baselines.



(a) Query on 300GB is CPU-bound. (b) Query on 1TB is IO-bound.

Figure 8: CPU utilization over time for TPC-DS query 89 on `r4.xlarge` cluster with *r*-SSD. For this query, performance with a small dataset is not indicative of performance with a larger dataset. Selecta detects difference in average iowait percentage (blue dotted line).

lecta's accuracy when recommending configurations for applications with large datasets solely based on profiling runs of the application with a smaller dataset. The solid lines show accuracy when Selecta re-profiles applications with large datasets to make predictions. For approximately 8% of applications, profiling runs with small datasets are not sufficient indicators of performance with large datasets.

We find that in cases where the performance with a small dataset is not indicative of performance with a large dataset, the relationship between compute and I/O intensity of the application is affected by the dataset size. As described in §3.3, Selecta detects these situations by comparing CPU utilization statistics for the small and large dataset runs. Figure 8 shows an example of a workload for which small dataset performance is not indicative of performance with a larger dataset. We use the Intel Performance Analysis Tool to record and plot CPU utilization [34]. When the average iowait percentage for the duration of the run changes significantly between the large and small profiling runs on the reference configuration, it is generally best to profile the application on the reference configurations and treat it as a new application.

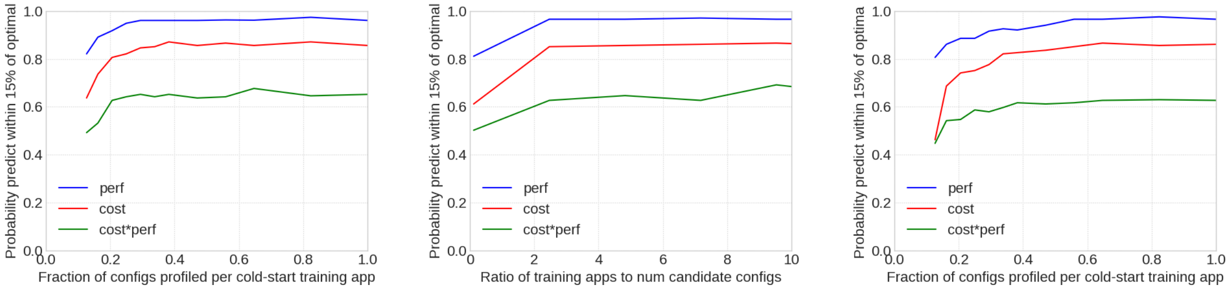
4.4 Sensitivity Analysis

We perform a sensitivity analysis to determine input matrix density requirements for accurate predictions. We look at both the density of matrix rows (i.e., the percentage of candidate configurations that training applications are profiled on) and the density of matrix columns (i.e., the number of training applications used). We also discuss sensitivity to the choice of reference configurations.

Figure 9a shows how Selecta's accuracy for performance, cost and cost*performance objectives varies as a function of input matrix density. Assuming 203 training applications have accumulated in the system over time, we show that, on average across target applications, rows only need to be approximately 20 to 30% dense for Selecta to achieve sufficient accuracy. This means that at steady state, users should profile training applications on about 20-30% of the candidate configurations (including reference configurations). Profiling additional configurations has diminishing returns.

Next, we consider a cold start situation in which a user wants to jump start the system by profiling a limited set of training applications across all candidate configurations. Figure 9b shows the number of training applications required to achieve desired accuracy. Here, for each target application testing round, we take the 203 training applications we have and randomly remove a fraction of the rows (training applications). We ensure to drop the row corresponding to the different dataset scale factor run of the target application, to ensure Selecta's accuracy does not depend on a training application directly related to the target application. Since the number of training applications required to achieve desirable accuracy depends on the size of the configuration space a user wishes to explore, the *x*-axis in Figure 9b represents the ratio of the number of training applications to the number of candidate configurations, *R*. We find that to jump start Selecta with dense training data from a cold start, users should provide 2.5× more training applications than the number of candidate configurations to achieve desirable accuracy. In our case, jump starting Selecta with more than $43 = \lceil 2.5 \times 17 \rceil$ training applications profiled on all 17 configurations reaches a point of diminishing returns.

Finally, we investigate whether, a cold start requires profile training applications on all configurations. We use *R*=2.5, which for 17 candidate configurations corresponds to using 43 training applications. Figure 9c plots accuracy as we vary the percentage of candidate configurations on which the training applications are profiled (including reference configurations, which we assume are always profiled). The figure shows that for a cold start, it is sufficient for users to profile the initial training applications on 40% to 60% of candidate configurations. As Selecta continues running and accumulates more training applications, the percentage of configura-



(a) Sensitivity to input matrix density in *steady state*: 20% density per row suffices for accurate predictions.

(b) Sensitivity to number of training applications, profiled on all configurations: 2.5× the number of configs suffices.

(c) Sensitivity to input matrix density for *cold start*: ∼50% density per row (training application) required.

Figure 9: Sensitivity analysis: accuracy as a function of input matrix density

tions users need to profile for training applications drops to 20-30% (this is the steady state result from Figure 9a).

We experimented with different reference configurations for Selecta. We find that accuracy is not very sensitive to the choice of references. We saw a slight benefit using references that have different VM and storage types. Although one reference configuration must remain fixed across all application runs since it is used to normalize performance, we found that the reference configuration used for the second profiling run could vary without significant impact on Selecta's accuracy.

5 Cloud Storage Insights

Our analysis of cloud configurations for data analytics reveals several insights for cloud storage configurations. We discuss key takeaways and their implications for future research on storage systems.

NVMe storage is performance and cost efficient for data analytics: We find that configurations with NVMe Flash tend to offer not only the best performance, but also, more surprisingly, the lowest cost. Although NVMe Flash is the most expensive type of storage per GB/hr, its high bandwidth allows applications to run significantly faster, reducing the overall job execution cost.

On average across applications, we observe that *l*-NVMe Flash reduces job completion time of applications by 27% compared to *r*-SSD and 75% compared to *r*-HDD. Although we did not consider *l*-SSD or *l*-HDD configurations in our evaluation, we validate that local versus remote access to HDD and SDD achieves similar performance since our instances have sufficient network bandwidth (up to 10 Gb/s) and modern networking adds little overhead on top of HDD and SSD access latency [8]. In contrast, a previous study of Spark applications by Ousterhout et al. concluded that optimizing or eliminating disk accesses can only reduce job completion time by a median of at most 19% [50]. We believe the main reason for the increased impact of storage on end-to-end application performance is due to the newer version of Spark we use in our study (v2.1.0 versus v1.2.1). Spark has evolved with numerous optimizations targeting CPU efficiency, such as cache-aware computations, code generation for expression evaluation, and serialization [17]. With ongoing work in optimizing the CPU cycles spent on data analytics computations, for example by optimizing the I/O processing path [66], we expect the choice of storage to be of even greater importance.

The need for flexible capacity and bandwidth allocation: Provisioning storage involves selecting the right capacity, bandwidth, and latency. Selecta uses statistics from Spark logs to determine capacity requirements and applies collaborative filtering to explore performance-cost trade-offs. However, the cost-efficiency of the storage configuration selected is limited by numerous constraints imposed by cloud providers. For example, for remote block storage volumes, the cloud provider imposes minimum capacity limits (e.g., 500 GB for *r*-HDD on AWS) and decides how data in the volume is mapped to physical devices, which directly affects storage throughput (e.g., HDD throughput is proportional to the number of spindles). A more important restriction is for local storage, such as *l*-NVMe, which is only available in fixed capacities attached to particular instance types. The fixed ratio between compute, memory and storage resources imposed by cloud vendors does not provide the right balance of resources for many of the applications we studied. For example the SQL equijoin query on two 64 GB tables saturates the IOPS of the 500 GB NVMe device on a `i3.xl` instance, but leaves half the capacity underutilized. Furthermore, local storage is ephemeral, meaning instances must be kept on to retain data on local devices. Thus, although we showed it is cost-efficient to store input/output and intermediate data on *l*-NVMe for the duration of a job, storing input/output files longer term on *l*-NVMe would dramatically increase cost compared to using remote storage volumes or an object storage system such as S3.

We make the case for a fast and flexible storage option in the cloud. Emerging trends in cloud computing, such as serverless computing offerings like AWS Lambda, Google Cloud Functions and Azure Functions, provide fine-grain, pay-per-use access to compute and memory resources [31, 7, 28, 46]. Currently, there is no option that allows for fine-grain capacity and bandwidth allocation of cloud storage with low latency and high bandwidth characteristics [41]. Although S3 provides pay-per-use storage with high scalability, high availability and relatively high bandwidth, we show that data analytics applications benefit from even higher throughput (i.e., NVMe Flash). S3 also incurs high latency, which we observed to be a major bottleneck for short-running SQL queries that read only a few megabytes of data.

Disaggregated NVMe is a promising option for fast and flexible cloud storage: Disaggregating NVMe Flash by enabling efficient access to the resource over the network is a promising option for fast and flexible cloud storage. Recent developments in hardware-assisted [49, 44] and software-only [40] techniques enable access to remote NVMe devices with low latency overheads over a wide range of network options, including commodity Ethernet networking with TCP/IP protocols. These techniques allow us to build disaggregated Flash storage that allows fine-grain capacity and IOPS allocation for analytics workloads and independent scaling of storage vs. compute resources. Applications would allocate capacity and bandwidth on demand from a large array of remotely accessible NVMe devices. In this setting, Selecta can help predict the right capacity and throughput requirements for each data stream in an analytics workload to guide the allocation of resources from a disaggregated Flash system.

There are several challenges in implementing flexible cloud storage based on disaggregated Flash. First, networking requirements can be high. Current NVMe devices on AWS achieve 500 MB/s to 4 GB/s sequential read bandwidth, depending on the capacity. Write throughput and random access bandwidth is also high. The networking infrastructure of cloud systems must be able to support a large number of instances accessing NVMe Flash remotely with the ability to burst to the maximum throughput of the storage devices. An additional challenge with sharing remote Flash devices is interference between read and write requests from different tenants [40, 61]. We observed several cases where separating input/output data and intermediate data on *r*-SSD (or S3) and *l*-NVMe, respectively, led to higher performance (and lower cost) than storing all data on *l*-NVMe. This occurred for jobs where large input data reads overlapped with large shuffle writes, such as for TPC-DS query 80 shown in Figure 1. A disaggregated Flash storage system must address interference using either scheduling approaches [40, 47, 61, 51, 60] or device-level isolation mechanisms [12, 54, 38]. Finally, the are interesting trade-offs in the interfaces used to expose disaggregated Flash (e.g., block storage, key-value storage, distributed file system, or other).

The need for end-to-end optimization: In our experiments, remote HDD storage performed poorly, despite its cost effectiveness for long-living input/output data and its ability to match the sequential bandwidth offered by SSD. Using the Linux `blktrace` tool [37] to analyze I/O requests at the block device layer, we found that although each Spark task reads/writes input/output data sequentially, streams from multiple tasks running on different cores interleave at the block device layer. Thus, the access stream seen by a remote HDD volume consists of approximately 60% random I/O operations, dramatically reducing performance compared to fully sequential I/O. This makes solutions with higher throughput for random accesses (e.g., using multiple HDDs devices or Flash storage) more appropriate for achieving high performance in data analytics. Increasing random I/O performance comes at a higher cost per unit time. In addition to building faster storage systems, we should attempt to optimize throughout the stack for sequential accesses when these accesses are available at the application level. Of course, there will always be workloads with intrinsically random access patterns that will not benefit from such optimizations.

6 Discussion

Our work focused on selecting storage configurations based on their performance and cost. Other important considerations include durability, availability, and consistency, particularly for long-term input/output data storage [42]. Developers may also prefer a particular storage API (e.g., POSIX files vs. object interface). Users can use these qualitative constraints to limit the storage space Selecta considers. Users may also choose different storage systems for high performance processing versus long term storage of important data.

Our study showed that separating input/output data and intermediate data uncovers a richer configuration space and allows for better customization of storage resources to the application requirements. We can further divide intermediate data into finer-grained streams such as shuffle data, broadcast data, and cached RDDs spilled from memory. Understanding the characteristics of these finer grain streams and how they should be mapped to storage options in the cloud may reveal further benefits.

Compression schemes offer an interesting trade-off between processing, networking, and storage requirements. In addition to compressing input/output files, systems like Spark allow compressing individual intermediate data streams using a variety of compression algorithms (lz4, lzf, and snappy) [64]. In future work, we plan to extend Selecta to consider compression options in addition to storage and instance configuration.

We used Selecta to optimize data analytics applications as they represent a common class of cloud workloads. Selecta's approach should be applicable to other data-intensive workloads too, as collaborative filtering does not make any specific assumptions about the application structure. In addition to considering other types of workloads, in future work, we will consider scenarios in which multiple workloads share cloud infrastructure. Delimitrou et al. have shown that collaborative filtering can classify application interference sensitivity (i.e., how much interference an application will cause to co-scheduled applications and how much interference it can tolerate itself) [22, 23]. We also believe Selecta's collaborative filtering approach can be extended to help configure isolation mechanisms that limit interference between workloads, particularly on shared storage devices like NVMe which exhibit dramatically different behavior as the read-write access patterns vary [40].

7 Related Work

Selecting cloud configurations: Several recent systems unearth near-optimal cloud configurations for target workloads. CherryPick uses Bayesian Optimization to build a performance model that is just accurate enough to distinguish near-optimal configurations [3]. Model input comes solely from profiling the target application across carefully selected configurations. Ernest predicts performance for different VM and cluster sizes, targeting machine learning analytics applications [69]. PARIS takes a hybrid online/offline approach, using random forests to predict application performance on various VM configurations based on features such as CPU utilization obtained from profiling [71]. These systems do not consider the vast storage configuration options in the cloud nor the heterogeneous data streams of analytics applications which can dramatically impact performance.

Resource allocation with collaborative filtering: Our approach for predicting performance is most similar to Quasar [23] and Paragon [22], which apply collaborative filtering to schedule incoming applications on shared clusters. ProteusTM [24] applies collaborative filtering to auto-tune a transactional memory system. While these systems consider resource heterogeneity, they focus on CPU and memory. While Selecta applies a similar modeling approach, our exploration of the cloud storage configuration space is novel and reveals important insights.

Automating storage configurations: Many previous systems provide storage configuration recommendations [9, 65, 2, 48, 4, 30, 39]. Our work analyzes the trade-offs between traditional block storage and object storage available in the cloud. We also considering how heterogeneous streams in data analytics applications should be mapped to heterogeneous storage options.

Analyzing performance of analytics frameworks: While previous studies analyze how CPU, memory, network and storage resources affect Spark performance [50, 68, 66, 43], our work is the first to evaluate the impact of new cloud storage options (e.g., NVMe Flash) and provide a tool to navigate the diverse storage configuration space.

Tuning application parameters: Previous work auto-tunes data analytics framework parameters such as the number of executors, JVM heap size, and compression schemes [32, 73, 72]. Our work is complementary. Users set application parameters and then run Selecta to obtain a near-optimal hardware configuration.

8 Conclusion

The large and increasing number of storage and compute options on cloud services makes configuring data analytics clusters for high performance and cost efficiency difficult. We presented Selecta, a tool that learns near-optimal configurations of compute and storage resources based on sparse training data collected across applications and candidate configurations. Requiring only two profiling runs of the target application, Selecta predicts near-optimal performance configurations with 94% probability and near-optimal cost configurations with 80% probability. Moreover, Selecta allowed us to analyze cloud storage options for data analytics and reveal important insights, including the cost benefits of NVMe Flash storage, the need for fine-gain allocation of storage capacity and bandwidth in the cloud, and the need for cross-layer storage optimizations. We believe that, as data-intensive workloads grow in complexity and cloud options for compute and storage increase, tools like Selecta will become increasingly useful for end users, systems researchers, and even cloud providers (e.g., for scheduling 'serverless' application code).

Acknowledgements

We thank our anonymous reviewers as well as Christina Delimitrou, Francisco Romero, and Neeraja Yadwadkar for their feedback. This work is supported by the Stanford Platform Lab, Samsung, Huawei and NSF grant CNS-1422088. Ana Klimovic is supported by a Stanford Graduate Fellowship and Microsoft Research PhD Fellowship.

References

[1] AGARWAL, S., KANDULA, S., BRUNO, N., WU, M.-C., STOICA, I., AND ZHOU, J. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012), NSDI'12, pp. 21–21.

[2] ALBRECHT, C., MERCHANT, A., STOKELY, M., WALIJI, M., LABELLE, F., COEHLO, N., SHI, X., AND SCHROCK, C. E. Janus: Optimal flash provisioning for cloud storage workloads. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (2013), USENIX ATC'13, pp. 91–102.

[3] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), pp. 469–482.

[4] ALVAREZ, G. A., BOROWSKY, E., GO, S., ROMER, T. H., BECKER-SZENDY, R., GOLDING, R., MERCHANT, A., SPASOJEVIC, M., VEITCH, A., AND WILKES, J. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst. 19*, 4 (Nov. 2001), 483–518.

[5] AMAZON. Amazon elastic block store (EBS). `https://aws.amazon.com/ebs`, 2017.

[6] AMAZON. Amazon simple storage service. `https://aws.amazon.com/s3`, 2017.

[7] AMAZON. AWS lambda. `https://aws.amazon.com/lambda`, 2017.

[8] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., AND STOICA, I. Disk-locality in datacenter computing considered irrelevant. In *Proc. of USENIX Hot Topics in Operating Systems* (2011), HotOS'13, pp. 12–12.

[9] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: Running circles around storage administration. In *Proc. of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02, USENIX Association.

[10] BELL, R., KOREN, Y., AND VOLINSKY, C. Modeling relationships at multiple scales to improve accuracy of large recommender systems. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2007), KDD '07, pp. 95–104.

[11] BELL, R. M., KOREN, Y., AND VOLINSKY, C. The BellKor 2008 Solution to the Netflix Prize. Tech. rep., 2008.

[12] BJØRLING, M., GONZALEZ, J., AND BONNET, P. Lightnvm: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (2017), pp. 359–374.

[13] BOTTOU, L. *Large-Scale Machine Learning with Stochastic Gradient Descent*. Physica-Verlag HD, 2010, pp. 177–186.

[14] CLOUDERA. How-to: Tune your apache spark jobs. `https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/`, 2015.

[15] CLOUDERA. Tuning spark applications. `https://www.cloudera.com/documentation/enterprise/5-9-x/topics/admin_spark_tuning.html`, 2017.

[16] CORPORATION, I. Intel Optane SSD DC P4800X Available Now on IBM Cloud. `https://www.ibm.com/blogs/bluemix/2017/08/intel-optane-ssd-dc-p4800x-available-now-ibm-cloud`, 2017.

[17] DATABRICKS. Project tungsten: Bringing apache spark closer to bare metal. `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`, 2015.

[18] DATABRICKS. AWS configurations for Spark. `https://docs.databricks.com/user-guide/clusters/aws-config.html#ebs-volumes`, 2016.

[19] DATABRICKS. Supported instance types. `https://databricks.com/product/pricing/instance-types`, 2016.

[20] DATABRICKS. Accelerating workflows on databricks. `https://databricks.com/blog/2017/10/06/accelerating-r-workflows-on-databricks.html`, 2017.

[21] DATABRICKS. Benchmarking big data sql platforms in the cloud. `https://databricks.com/blog/2017/07/12/benchmarking-big-data-sql-platforms-in-the-cloud.html`, 2017.

[22] DELIMITROU, C., AND KOZYRAKIS, C. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13, pp. 77–88.

[23] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS '14, pp. 127–144.

[24] DIDONA, D., DIEGUES, N., KERMARREC, A.-M., GUERRAOUI, R., NEVES, R., AND ROMANO, P. Proteustm: Abstraction meets performance in transactional memory. In *Proc. of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ASPLOS '16, pp. 757–771.

[25] FERGUSON, A. D., BODIK, P., KANDULA, S., BOUTIN, E., AND FONSECA, R. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys '12, pp. 99–112.

[26] FOUNDATION, A. S. Apache parquet. `https://parquet.apache.org/`, 2014.

[27] GHAZAL, A., RABL, T., HU, M., RAAB, F., POESS, M., CROLOTTE, A., AND JACOBSEN, H.-A. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD '13, pp. 1197–1208.

[28] GOOGLE. Cloud functions. `https://cloud.google.com/functions`, 2017.

[29] GOOGLE. Google compute engine persistent disk. `https://cloud.google.com/persistent-disk`, 2017.

[30] GULATI, A., SHANMUGANATHAN, G., AHMAD, I., WALDSPURGER, C., AND UYSAL, M. Pesto: Online storage performance management in virtualized datacenters. In *Proc. of the 2Nd ACM Symposium on Cloud Computing* (2011), SOCC '11, pp. 19:1–19:14.

[31] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[32] HERODOTOU, H., LIM, H., LUO, G., BORISOV, N., DONG, L., CETIN, F. B., AND BABU, S. Starfish: A self-tuning system for big data analytics. In *In CIDR* (2011), pp. 261–272.

[33] HUG, N. Surprise, a Python library for recommender systems. `http://surpriselib.com`, 2017.

[34] INTEL. Performance analysis tool (PAT). `https://github.com/intel-hadoop/PAT`, 2016.

[35] INTEL. Intel optane technology. `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html`, 2017.

[36] JALAPARTI, V., BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Bridging the tenant-provider gap in cloud services. In *Proceedings of the Third ACM Symposium on Cloud Computing* (2012), SoCC '12, pp. 10:1–10:14.

[37] JENS AXBOE, A. D. B., AND SCOTT, N. blktrace man page. `https://linux.die.net/man/8/blktrace`, 2006.

[38] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems* (2014), HotStorage'14, pp. 13–13.

[39] KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. Designing for disasters. In *Proc. of the 3rd USENIX Conference on File and Storage Technologies* (2004), FAST '04, pp. 59–62.

[40] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. ReFlex: Remote flash == local flash. *SIGPLAN Not. 52*, 4 (Apr. 2017), 345–359.

[41] KLIMOVIC, A., WANG, Y., KOZYRAKIS, C., STUEDI, P., PFEFFERLE, J., AND TRIVEDI, A. Understanding ephemeral storage for serverless analytics. In *Proc. of the USENIX Annual Technical Conference* (2018), ATC'18.

[42] KOVACS, G. EBS, EFS, or Amazon S3: which is the best cloud storage system for you? `https://cloud.netapp.com/blog/ebs-efs-amazons3-best-cloud-storage-system`, 2017.

[43] LI, H., GHODSI, A., ZAHARIA, M., SHENKER, S., AND STOICA, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), SOCC '14, pp. 6:1–6:15.

[44] METZ, J., HUFFMAN, A., SARDELLA, S., AND MINTRUN, D. The performance impact of NVM Express and NVM Express over Fabrics. `http://www.nvmexpress.org/wp-content/uploads/NVMe-Webcast-Slides-20141111-Final.pdf`, 2015.

[45] MICROSOFT. Azure files. `https://azure.microsoft.com/en-us/services/storage/files`, 2017.

[46] MICROSOFT. Azure functions. `https://azure.microsoft.com/en-us/services/functions`, 2018.

[47] NANAVATI, M., WIRES, J., AND WARFIELD, A. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (2017), pp. 17–33.

[48] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating server storage to ssds: Analysis of tradeoffs. In *Proceedings of the 4th ACM European Conference on Computer Systems* (2009), EuroSys '09, pp. 145–158.

[49] NVM EXPRESS INC. NVM Express over Fabrics Revision 1.0 . `http://www.nvmexpress.org/wp-content/uploads/NVMe_over_Fabrics_1_0_Gold_20160605.pdf`, 2016.

[50] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (2015), NSDI'15, pp. 293–307.

[51] PARK, S., AND SHEN, K. FIOS: a fair, efficient flash I/O scheduler. In *Proc. of USENIX File and Storage Technologies* (2012), FAST'12, p. 13.

[52] PERFORMANCE IO RESEARCH GROUP AT IBM RESEARCH ZURICH, H. Example terasort program. `https://github.com/zrlio/crail-spark-terasort`, 2017.

[53] PERFORMANCE IO RESEARCH GROUP AT IBM RESEARCH ZURICH, H. Spark sql benchmarks. `https://github.com/zrlio/sql-benchmarks`, 2017.

[54] PETERSEN, C., AND HUFFMAN, A. Solving latency challenges with NVM express SSDs at scale. `https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170809_SIT6_Petersen.pdf`, 2017.

[55] POPESCU, A. D., ERCEGOVAC, V., BALMIN, A., BRANCO, M., AND AILAMAKI, A. Same Queries, Different Data: Can we Predict Query Performance? In *Proceedings of the 7th International Workshop on Self Managing Database Systems* (2012).

[56] RICCI, F., ROKACH, L., SHAPIRA, B., AND KANTOR, P. B. *Recommender Systems Handbook*, 1st ed. Springer-Verlag New York, Inc., 2010.

[57] SALAKHUTDINOV, R., AND MNIH, A. Probabilistic matrix factorization. In *Proceedings of the 20th International Conference on Neural Information Processing Systems* (2007), NIPS'07, pp. 1257–1264.

[58] SAMMUT, C., AND WEBB, G. I., Eds. *Leave-One-Out Cross-Validation*. Springer US, 2010, pp. 600–601.

[59] SCHAD, J., DITTRICH, J., AND QUIANÉ-RUIZ, J.-A. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 460–471.

[60] SHEN, K., AND PARK, S. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *Proc. of USENIX Annual Technical Conference* (2013), ATC'13, USENIX, pp. 67–78.

[61] SHUE, D., AND FREEDMAN, M. J. From application requests to virtual IOPs: provisioned key-value storage with Libra. In *Proc. of European Conference on Computer Systems* (2014), EuroSys'14, pp. 17:1–17:14.

[62] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In *Proc. of IEEE Mass Storage Systems and Technologies* (2010), MSST '10, IEEE Computer Society, pp. 1–10.

[63] SPARK, A. Monitoring and instrumentation. `https://spark.apache.org/docs/latest/monitoring.html`, 2017.

[64] SPARK, A. Spark configuration. `https://spark.apache.org/docs/latest/configuration.html`, 2017.

[65] Strunk, J. D., Thereska, E., Faloutsos, C., and Ganger, G. R. Using utility to provision storage systems. In *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA* (2008), pp. 313–328.

[66] Stuedi, P., Trivedi, A., Pfefferle, J., Stoica, R., Metzler, B., Ioannou, N., and Koltsidas, I. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Eng. Bull. 40*, 1 (2017), 38–49.

[67] TPC, T. P. P. C. TPC-DS is a Decision Support Benchmark. `http://www.tpc.org/tpcds/`, 2017.

[68] Trivedi, A., Stuedi, P., Pfefferle, J., Stoica, R., Metzler, B., Koltsidas, I., and Ioannou, N. On the [ir]relevance of network performance for data processing. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing* (2016), HotCloud'16, pp. 126–131.

[69] Venkataraman, S., Yang, Z., Franklin, M., Recht, B., and Stoica, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), pp. 363–378.

[70] Vitter, J. S. Random sampling with a reservoir. *ACM Trans. Math. Softw. 11*, 1 (Mar. 1985), 37–57.

[71] Yadwadkar, N. J., Hariharan, B., Gonzalez, J. E., Smith, B., and Katz, R. H. Selecting the *best* VM across multiple public clouds: a data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), SOCC'17, pp. 452–465.

[72] Yeh, C. C., Zhou, J., Chang, S. A., Lin, X. Y., Sun, Y., and Huang, S. K. Bigexplorer: A configuration recommendation system for big data platform. In *2016 Conference on Technologies and Applications of Artificial Intelligence (TAAI)* (Nov 2016), pp. 228–234.

[73] Yigitbasi, N., Willke, T. L., Liao, G., and Epema, D. Towards machine learning-based autotuning of mapreduce. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems* (Aug 2013), pp. 11–20.

[74] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (2010), HotCloud'10, pp. 10–10.

[75] Zhou, P., Ruan, Z., Fang, Z., Shand, M., Roazen, D., and Cong, J. Doppio: I/o-aware performance analysis, modeling and optimization for in-memory computing framework.

Remote regions: a simple abstraction for remote memory

Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi
Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam
Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, Michael Wei
VMware

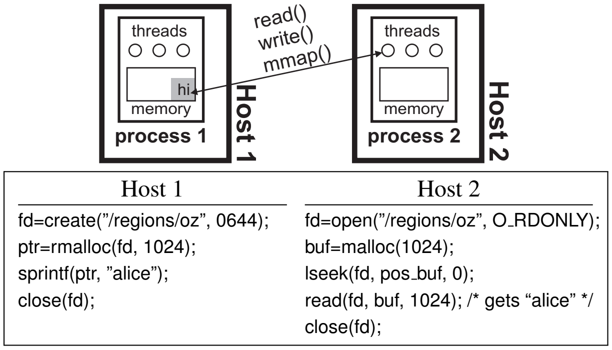
Abstract

We propose an intuitive abstraction for a process to export its memory to remote hosts, and to access the memory exported by others. This abstraction provides a simpler interface to RDMA and other remote memory technologies compared to the existing *verbs* interface. The key idea is that a process can export parts of its memory as files, called *remote regions*, that can be accessed through the usual file system operations (read, write, memory map, etc). We built this abstraction in the Linux kernel, and evaluated it. We show that remote regions are easy to use and perform close to RDMA. We demonstrate it via micro-benchmarks and by adapting two in-memory single-host applications to use remote memory: R and Metis. With R, using remote regions requires no changes to the code and allows R to run with remote memory that exceeds the physical memory of a host. With Metis, the modifications amount to ≈100 lines of code and they allow Metis to scale its performance across 8 hosts.

1 Introduction

Remote memory allows a process to read and write the memory of another process in a different host. This is an exciting idea whose time has come [1]. Remote memory is available now, using RDMA technology over Infiniband or Ethernet [49, 29], and other new technologies are emerging [28, 24, 47]. Many applications are being redesigned to use remote memory (key-value storage systems [43, 14, 30, 15], database systems [50, 6, 64], map-reduce [39], etc).

Unfortunately, remote memory faces two problems now. First, it has no standard interface. Current technology uses the RDMA verbs interface, but new hardware such as Gen-Z and OpenCAPI will have their own interfaces to control mapping, access, etc. Even RDMA is still changing with key innovations, such as DCT [18], that are offered in some implementations but not others. Second, remote memory today is hard to use. With RDMA, even the simplest program to access some data from a remote host requires a complex ritual: code is required to initialize contexts, register memory, establish RDMA connections, create queue-pairs, associate them with connections, transition the queues through various states, exchange RDMA keys, post commands on queues, and poll the queues for completions [7]. Furthermore, RDMA lacks naming and location services that applications need, forcing them to reimplement this functionality every time.



Host 1	Host 2
fd=create("/regions/oz", 0644);	fd=open("/regions/oz", O_RDONLY);
ptr=rmalloc(fd, 1024);	buf=malloc(1024);
sprintf(ptr, "alice");	lseek(fd, pos_buf, 0);
close(fd);	read(fd, buf, 1024); /* gets "alice" */
	close(fd);

Figure 1: Using regions, Host 1 creates a region named *oz* in REGIONFS, allocates a buffer in the region, and populates the buffer. Host 2 then reads host 1's string, similar to an RDMA-read operation, except that developers need not program with RDMA directly (which is complex).

In this paper, we propose a simple idea: to use files as the interface to remote memory, shedding the complexity of RDMA and providing a standard for new technologies. In particular, we propose *remote regions* or, in short, *regions*. With regions, a process exports parts of its memory as *files* in REGIONFS, a file system that a remote host can then access using the usual file operations (read, write, memory map, etc). In addition to a simple interface, regions draw features from file systems to provide functionality lacking in RDMA: name space, timestamps, access control, etc (§4).

Regions are simple because they replace low-level RDMA mechanisms with high-level controls that are operated through a familiar interface. Figure 1 shows how easily a host can use regions to read data in the memory of another host. By contrast, equivalent RDMA logic takes around 300 lines of hard-to-understand code [7].

The main challenge in designing regions is to find the right balance between elegance, expressiveness, and efficiency, while overcoming the limitations of the hardware. To find this balance, we address questions of file semantics, memory allocation, data sharing, memory mapping, page fault preemption, security, data-metadata separation, caching, cache coherence, and sharing granularity, while addressing RDMA limits on memory registration, connections, and keys. The current implementation of regions targets RDMA, but we believe region's interface will be applicable to new upcoming remote memory technologies [24, 28], providing a common abstraction with which applications can be written in a portable fashion across these new technologies.

We have built regions using RDMA in the Linux kernel v4.8, and we evaluated their cost on a cluster of 8 machines with RoCE. Using microbenchmarks, we see that accessing data via regions is reasonably close to RDMA. We have used regions to extend two applications, R [48] and Metis [41], to use remote memory. R is a system for statistical processing of data, while Metis is an in-memory implementation of map-reduce running on a single host. We use regions to adapt R to operate on large data sets in remote memory exceeding the available local memory capacity. We do this by using R's ff package to store objects in memory-mapped files, and placing these files in REGIONFS. We also use regions to produce a distributed version of Metis that runs across many hosts, sharing in-memory data. This change required only 82 lines of code, and allows Metis to scale to 8 hosts, giving it more memory and improving performance by 3.5× compared to a single host.

2 Related work

Same interface, different goal. The file interface is often used for *remote storage*, where the main goal is to provide durable storage capacity. Several such systems use RDMA to improve performance, such as Octopus [40], Crail [57], Ceph [11], and GlusterFS [25]. DAFS [13] is a file system protocol for RDMA, while [60] is a proposal to run NFS over RDMA. The file interface can also be used to manage large local memories [58]. All of the above works rely on a file interface but have a different goal from our goal of accessing the memory of remote applications.

Different interface, same goal. Prior work provides remote memory with a different interface. LITE [61] provides a kernel interface that offers more flexible protection, and better scalability and isolation than verbs on RDMA. There is much work in distributed shared memory (DSM) (e.g., [9, 33, 46, 2, 51, 53, 56, 8]) including recent work on persistence using non-volatile memory and replication [54]. FaRM [14, 15] provides transactions over RDMA with lock-free reads. All these systems provide a simpler interface than RDMA, but they do not support the well-known file interface, which has many advantages (§4).

Different interface, different goal. Many systems provide remote storage with an interface other than files, including key-value stores (e.g., [43, 14, 30, 15]), Linda tuples [10], distributed objects (e.g., [63, 27, 5]), and database systems. These systems offer a different abstraction from regions. For example, key-value stores provide GETs and PUTs on key-value pairs; Linda provides a tuple interface; distributed objects require applications to declare and manipulate the objects provided by the framework; and database systems use SQL.

Remote memory applications. Several works have proposed replacing disks with remote memory as a faster target for swapping or paging (e.g., [23, 12, 34, 21, 31, 22, 17]). CacheDM [36] uses remote memory as a cache for a network file system, while Infiniswap [26] uses remote memory as a cache for a local swap/paging device. Several of these applications are built with RDMA; they might have been simpler to develop with regions.

New hardware. Disaggregated memory proposes a new system architecture that detaches memory from machines and places it on a common fabric. The work includes academic papers [26, 37, 20, 4, 23, 45] and upcoming technologies to support it, such as Gen-Z [24] and Omni-Path [28]. Regions could provide an elegant interface to disaggregated memory, though the implementation of regions will differ from the RDMA implementation we give (that will depend on the details of these technologies, which are still work in progress).

3 Assumptions, goals, and motivation

We assume machines are connected to a network with low latency, high bandwidth, and reliable connectivity—such as, for example, machines in a few racks in a data center. We assume a single administrative, trust, and fault domain. We consider deployments with a couple to tens of machines. While some companies have large deployments with thousands of machines, the vast bulk of our customers are enterprises with deployments of 100 or fewer machines in a private facility, and that is our target environment. Network partitions are rare and, when they do occur, it is reasonable for the system to pause as the rest of the system will be unavailable anyways (e.g., network file systems and other servers are unreachable).

Our goal is to provide abstractions for applications to access the memory of other applications across the network. Currently, the standard way to do that is to employ one-sided read and write operations using the verbs library (libibverbs [35]). This interface has three issues that we want to overcome:

- *Complexity.* As we mentioned, verbs operations are complex, and we seek simple and intuitive alternatives.
- *Dependency on existing technology.* There are other remote memory technologies under development other than RDMA, such as Omni-Path [28] and Gen-Z [24]. We would like to find high-level abstractions so that applications can be portable across these technologies.
- *Resource limitations.* RDMA has limitations on resources at the network adapter, such as limited cache sizes for connections and memory translations [14]. We

want to design abstractions that can hide or overcome these limitations without concerning applications.

We expect a simpler interface to have performance costs but want them to be reasonable and we certainly want to understand them, much like a developer needs to understand the cost of other high-level features, such as garbage collection, lambdas, etc.

4 Why files?

By using a file interface, regions get many benefits:

- *Well-known.* All developers know files.
- *Utilities.* The file interface inherits a vast repertoire of utilities: editors, backup, grep, find, cp, cat, sed, awk, etc. Regions allow these to be used with remote memory.
- *Language support.* Most of the functionality of regions is in REGIONFS, and all major programming languages support files. There is only a small library (with synchronization and stub functions) that needs to be ported to a given language.
- *Interposition support.* There are many tools to interpose on file system calls, for tracing, debugging, auditing, and profiling (e.g., DFSTrace [44]). These tools all work with REGIONFS.
- *Name space.* Directories and files make it easy to find and organize data across applications in the network.
- *Users and access permissions.* Applications can use the notion of users from the operating system combined with access permissions to control who has access.

We get these benefits for free because the file interface is well matched to our problem. In contrast, other interfaces to remote memory, such as RDMA, provide none of these benefits.

5 The regions abstraction

We now explain how regions appear to users as an abstraction, and we explain how we arrived at this abstraction. We show how to provide the abstraction in §6.

In its simplest form, a (remote) region is a logically contiguous part of the memory of a process, called the *owner* process. The owner creates a region like a file, and can operate on it by memory mapping, reading, writing, or allocating variables using a special *rmalloc* function (§5.2); these operations refer to data in local memory. Processes in other hosts can also perform these operations, to access data in the memory of the owner.

5.1 Basic functions

Regions provide a file system called REGIONFS mounted in a known location, such as /regions. Each file in REGIONFS is a region stored in memory, either locally or remotely. REGIONFS supports the usual file operations (e.g., creat, unlink, open, close, read, write, chmod, stat) in addition to mmap (§5.7). By default, a region disappears when its creator process terminates or crashes (accessing it results in an I/O error).

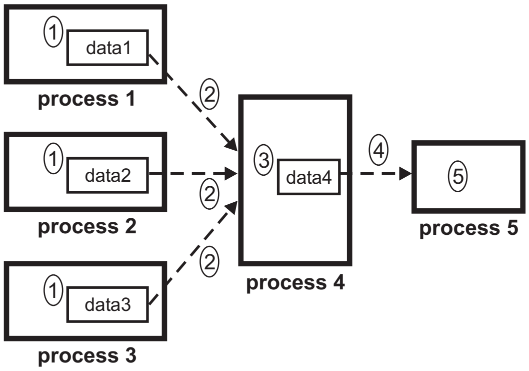


Figure 2: Example of the use of regions on a map-reduce style of computation. (1) Three mapper processes in different hosts create a region each, run some computation, and store the results in their region; (2) a fourth reducer process reads the data in the regions, and (3) creates a region and writes the result there; (4) a fifth process reads that region and (5) produces a graph in the display of a user.

A directory in REGIONFS is not a region but organizes regions, much like regular file systems; but unlike regular file systems, directories carry some special extended attributes that regions inherit upon creation (§5.4).

5.2 Memory allocation

An application often dynamically allocates and destroys many buffers in its lifetime. Rather than creating/deleting a region for each buffer, applications can dynamically allocate/free buffers *within* a region, using these functions:

```
void *rmalloc(int regionfd, size_t len)
int rfree(void *ptr)
```

where *regionfd* is a descriptor for a region open in write mode. Calling *rmalloc* is faster than creating a region: the former executes entirely in memory, while the latter requires contacting a metadata manager over the network (§6.6). In fact, an application might create just one region and then allocate its buffers within that region.

5.3 Example of usage

We illustrate the use of a region with an example with five processes that run a map-reduce style of computation (Figure 2). In existing map-reduce systems, processes exchange data using a distributed file system such as the Hadoop Distributed File System (HDFS) [3]. With regions, processes can exchange data directly in memory, as with RDMA, but with the simplicity of using files. Also, this is distinct from using an RDMA-enabled file system (e.g., [40, 11, 25]), where processes store data in a storage server and use RDMA to access the server; with regions, processes can directly export data in *their* memory and read data from the memory of other processes.

Attribute	Type	Description
OWNERPID	pid_t	pid of process owning region
PERSISTENT	bool	keep region when process ends (§5.5)
MULTIHOSTED	bool	store region across many hosts (§5.6)
FIXVADDR	uint64_t	fixed virtual address (§5.7)
ONEWRITER	bool	only one process can open for writing
HOSTALLOC	ip list	hosts storing region if MULTIHOSTED (§5.6)

Figure 3: Region-specific attributes.

5.4 Region attributes

Beyond the attributes of a typical file (access bits, uid, gid, etc), each region has some additional region-specific metadata that determine certain behaviors (Figure 3). The owner indicates the process who created the region; this is different from the owner uid of a region/file, which is a user. When this process ends, the region is automatically deleted unless the PERSISTENT attribute is set (§5.5). A region gets a fixed virtual address across hosts if FIXVADDR is set (§5.7). A region can be opened for writing by at most one process if ONEWRITER is set; this is enforced across hosts. When a region grows in size, new memory is typically allocated from the host of the owner, but it is possible to allocate it from remote hosts as well if MULTIHOSTED is set, in which case HOSTALLOC indicates the hosts to allocate from (§5.6).

Applications set these attributes when the region is created. Since we use the standard creat() call to create regions, which cannot specify attributes, we define an additional function

```
int rsetdefaultattr(int attr, char *val, int len) /* returns error flag */
```

that sets the default attributes of new regions for the calling thread.

5.5 Persistent regions

By default, a region is backed by the memory of a process. If the process terminates, its memory is deallocated and the region is automatically deleted. This could be undesirable in some cases: the process might wish to leave the data in memory for a short while until it is consumed by another process. One solution to this problem is for the process to defer its termination until its data has been consumed. This solution is complex because it requires the process to coordinate with other applications.

We provide a simpler solution: to retain the region contents after the process terminates. Upon termination, a process releases its memory but not the region. We call such regions *persistent* regions. Persistent regions should be deleted by the consuming process later. They are also deleted when the host reboots. To create a persistent region, a process sets the attribute PERSISTENT.

5.6 Multi-hosted regions

A multi-hosted region is a special type of region that is stored across many hosts. These regions can store large data that exceed the physical memory of any single host.

To create a multi-hosted region, a process sets attribute MULTIHOSTED and optionally chooses the hosts where the region will be allocated via attribute HOSTALLOC (§5.4); if this is not set, the default is to use all hosts.

5.7 Memory mapping

Processes can memory map a region using mmap(), so that the region can be accessed by memory operations instead of read() and write(). The function returns a pointer where the region is mapped. If a region is created with the FIXVADDR attribute, it is given a fixed virtual address [54]: it always maps to that address, no matter which process or host maps the region. This ensures that pointers to data in regions remain valid across hosts, allowing regions to store dynamic data structures and other data that require indirection. To implement this feature, we reserve virtual addresses across the cluster (§6.10).

5.8 Performance enhancing functions

Memory mapping of a region on a remote host is implemented using page faults. Page faults have two causes: (1) when a process first accesses a page, to fetch the page; (2) when the process first writes to the page, to mark it dirty. If the first access is a write, one page fault both fetches and marks it dirty. Because page faults are expensive, we provide two ways to prevent them: prefetch and mark-dirty. With prefetch, applications request the system to fetch pages immediately, by calling

```
int rprefetch(void *addr, size_t len, bool sync) /* ret: error flag */
```

which prefetches data in a region starting at *addr* with length *len*; if *sync* is set, it waits until the data it fetched. To avoid page faults due to writes, applications can request the system to mark the page dirty, by calling

```
int rmarkdirty(void *addr, size_t len, bool zero) /* ret: err flag */
```

before writing to a page. If parameter *zero* is true, this function zeroes the pages without reading their contents. This is useful to avoid the overhead of a read-modify-write cycle if the application intends to completely overwrite the pages (see §8.3).

Function rprefetch is just an optimization that does not change application semantics. Function rmarkdirty is also an optimization when parameter *zero* is false; if *zero* is true, rmarkdirty is equivalent to bzero().

5.9 Synchronization

When using regions, one might need to synchronize processes across hosts (e.g., to share data, as in §5.3). We provide several distributed synchronization primitives: barriers, mutexes, and door bells (Figure 4). These are

Function	Description
rbarrier_init(name, n)	Create barrier for *n* callers
rbarrier_wait(name)	Wait for barrier
rmutex_init(name)	Create mutex
rmutex_lock	Acquire mutex
rmutex_unlock	Release mutex
rbell_init(name)	Create door bell
rbell_ring(name)	Increment bell value
rbell_wait(name)	Wait for new value, return it
rdelete(name)	Deallocate

Figure 4: Available synchronization primitives.

offered in a user library, since a file system has no such functionality. A barrier has a parameter *n* and the caller blocks until the barrier has been called at least *n* times. This serves to synchronize a group of processes. A mutex ensures at most one caller gets the mutex at once. A door bell has an initial value 0; the ring function increments it; the wait function waits for it to be incremented since its last call, returning the current value.

5.10 Caching

When a process uses a region of a different host, the system locally maintains a page cache of data that has been recently read or that has been modified. The cache is a write-back cache (modifications are propagated back to the region in the background). The system does not provide cache coherence, because it is too expensive; rather applications can obtain coherence at the moments of their choice by explicitly using two mechanisms, flushing and clearing caches:

```
int msync(void *addr, size_t len, int flags) /* returns error flag */
int rclearcache(void *addr, size_t len) /* return error flag */
```

where addr is the address within one of the open regions. Flushing [msync] causes dirty pages to be written back to the region, so the owner can observe the modified data. Clearing pages [rclearcache] removes them from the cache, so that the calling process subsequently obtains fresh data from the owner. These functions produce an effect only at a process remote to the region or for multi-hosted regions, as the owner of single-hosted region does not have a cache. After clearing a page, a process might invoke rprefetch() to avoid a page fault (§5.8).

Processes sharing a region must follow some discipline on how to use these functions to avoid data corruption. We propose a simple and effective scheme in the next section.

5.11 Sharing data

To correctly share data, processes must flush and clear their caches carefully. Doing so is not easy in general, but we now describe a simple scheme that works well in the use cases that regions are designed for. To explain how this is done, we broadly classify sharing of data alongside two dimensions.

Type	Regions	RDMA
Owner-remote	Owner writes to region and remote process reads, or remote process writes to region and owner reads	One process writes locally and another RDMA-reads, or one process RDMA-writes and another reads locally
Remote-remote	A remote process writes to region and a remote process reads from region	One process RDMA-writes to third party's memory and another RDMA-reads

Figure 5: Two patterns of sharing data between processes in different hosts using regions and the analogue using RDMA.

The first dimension is who participates in the sharing relative to who owns the data. There are two possibilities: owner-remote sharing and remote-remote sharing (Figure 5). With owner-remote sharing, one of the processes sharing owns the region or the memory. With remote-remote, the process that owns the region or memory is a third party. Owner-remote sharing is simpler to deal with, because there is only one cache involved (the cache of the remote process), while remote-remote sharing involves two caches, one for each remote process.

The second dimension is what we call the *granularity of sharing*. With fine-grained sharing, processes interleave their execution often and share small bits of data (e.g., one or a few variables) at a time, with frequent coordination. For example, in a mutual exclusion algorithm, two processes frequently read and write common variables containing the state of flags or counters, often changing the role of who reads and writes the shared information. With coarse-grained sharing, one process produces a large chunk of data before another process consumes it; for example, in the map-reduce computation of Figure 2, the mappers produce large outputs that are later consumed by the reducer.

We anticipate that regions will be used for both owner-remote and remote-remote sharing alongside the first dimension, but only for coarse-grained sharing alongside the second dimension, because fine-grained sharing over the network is generally too costly. Coarse-grained sharing does not require the cache to be coherent very often: it suffices to be coherent in the instant after the producer has finished writing and before the consumer starts reading. Accordingly, processes can flush or clear their caches at that moment, as follows. With owner-remote sharing, the remote process either flushes or clear its cache, depending on whether it is producing or consuming data. With remote-remote sharing, the remote process that produces data flushes its cache, while the remote process that consumes data clears its cache.

5.12 Pseudo file system

Regions have more metadata than files. We expose this metadata to users in a pseudo file system /proc/regions,

File name	Description
hosts	List of hosts using regions
memusage	Aggregate memory usage of regions hosted locally
pools	List of pools (§6.7) with used/free space and daemon's logical address
daemon	pid of daemon process (§6.4)
procs	pid of local processes using regions
files/*pathname*	Metadata of region in *pathname*: vaddr, maximum size, pool, attributes

Figure 6: Region metadata stored in pseudo file system /proc/regions.

accessible only by root. Available Information includes local memory usage of regions, a list of local pools, and the local processes using regions (Figure 6). Moreover, for each region *r*, /proc/regions/files/*r* indicates the region's fixed virtual address, maximum size, pools from which memory is allocated, and attributes.

5.13 Limitations

Regions have a limitation: a process cannot use them to export data in its stack or static variables, because the process must allocate data in regions using rmalloc. However, these limitations may not matter: it is easy to change static variables to heap variables, and it is probably a bad idea for applications to export data in the stack, since that data disappears when its call frame is deleted.

6 Realizing regions using RDMA

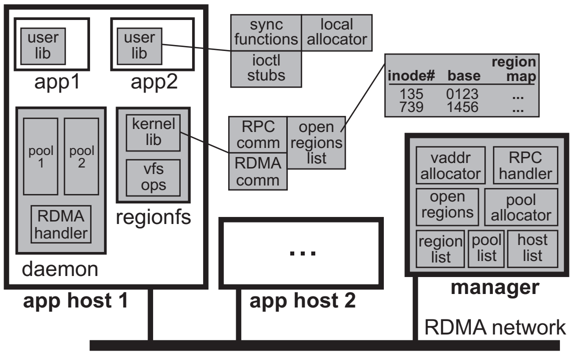
We now describe how we realize regions using RDMA. While the design is centered around RDMA, we expect that its key ideas will be applicable to future disaggregated memory hardware.

6.1 Basic architecture

Figure 7 shows the architecture of regions. There are four main logical components: REGIONFS file system, user library, daemon, and manager. Broadly, the REGIONFS file system component implements the VFS kernel operations required of a file system, while the user library implements synchronization and performance-enhancing functions. The first module is instantiated once per host; the second, once per application. The daemon (one per host) allocates and maintains large pools of memory in which regions are allocated, and shares these pools with both local and remote processes. The manager provides the control plane, handling every file system operation except reading and writing data. The manager has one instance but it is replicated for high availability using standard mechanisms, such as Paxos state machine replication [32, 52]. We provide more details in the next sections.

6.2 RegionFS file system component

This component is a kernel module that implements the file system for region, with functionality to drive the execution of file system operations, coordinating with the manager and the other hosts' daemons. The module keeps an important data structure, the open region list, which tracks all regions that the application has opened, with their virtual addresses, and map for locating the data within the region. We detail the VFS operations in §6.6 after some more background, but they fall into two categories: Data operations (read, write, readpage, etc) execute locally or over RDMA, depending on where a region resides. Metadata operations (directories, file attributes, etc) are similar to the implementation of a network file system (e.g., to create a directory, it calls the manager, which then records information about the directory).



Module	Description	Section
regionfs	file system for regions	§6.2
user lib	user-level file system	§6.3
daemon	holds and exports memory pools	§6.4
manager	handles control operations	§6.5
sync funcs	barrier, mutex, doorbell	§6.3
local alloc	kmalloc	§6.3
ioctl stubs	functions without VFS analogues	§6.3
kernel lib	communication library	§6.12
RPC comm	for communicating with manager	§6.12
RDMA comm	for communicating with daemons	§6.12
open regions list	list of regions that client has opened	§6.2
region map	tracks where a region is stored	§6.8
vfs ops	VFS interface to file system	§6.6
RDMA handler	accepts RDMA connections	§6.12
vaddr allocator	allocates virtual addresses	§6.10
RPC handler	handles requests from clients	§6.12
open regions	all open regions in the system	§6.5
pool allocator	allocates cluster memory	§6.9
region list	all regions in the system	§6.5
pool list	all pools in the system	§6.5
host list	keep track of hosts	§6.5

Figure 7: Architecture. Region components are in gray. The figure shows two application hosts, but we expect a few dozens of them. There is a single manager, and it is replicated for fault tolerance. The manager is involved only in infrequent control operations, staying out of the performance-critical data path.

6.3 User library component

The user library provides the synchronization functions (§5.9), an allocator for rmalloc (§5.2), and ioctl stubs. The synchronization functions issue an RPC to the manager, which implements the actual functionality. The RPC call blocks until the synchronization occurs (e.g., a barrier gets all its participants). For the rmalloc allocator, we memory map the region (if it has not been mapped already) and organize the space using a buddy allocator. The allocator uses a magic number at the beginning of the region to know if the allocator structures must be initialized. The ioctl stubs provide region-specific functions without a corresponding VFS operation (rsetdefaultattr (§5.4), rprefetch (§5.8), rmarkdirty (§5.8), rclearcache (§5.10)). The stubs translate these functions into ioctl's that get handled by VFS.

6.4 The daemon

The daemon serves three purposes. First, it overcomes resource limitations of the RDMA network adapter, which cannot keep many connections or export many buffers because its internal cache is small [14]. To address these problems, the daemon allocates big pools of physical memory (§6.7) and then allocates regions within these pools. Thus, a host exports a few pools (rather than many regions) and a remote process can connect just with the daemon to access the data of all applications in the host (instead of connecting to each application). Second, the daemon allows a host to offer memory to multi-hosted regions (§5.6) even if the host has no running applications. Third, the daemon supports persistent regions (§5.5) by holding the region's data when a process terminates.

6.5 Manager

A central manager handles all control operations: creation, opening, closing, deletion, and memory-mapping of regions; file system metadata operations (create and delete directories, set and get inode attributes); allocation of memory for regions; and allocation of global virtual addresses. To do that, the manager keeps track of the hosts in the system, file system metadata (inodes and directory contents), memory usage of all pools at each host, allocation of regions, allocation of virtual addresses, and list of all open regions. The manager is not involved in reading and writing data in regions—the performance critical operations—so it is not a bottleneck. However, a larger system might require distributing the manager.

6.6 VFS operations

To open a region [open()], the client calls the manager; if the region exists, the manager returns its starting virtual address and region map (§6.8); the client adds the region to its open regions list and stores its virtual address and region map. To create a region, the client calls the manager to check if the region already exists, to preallocate an initial set of pages to it, to allocate virtual addresses (§6.10), and to return the starting virtual address and region map, which the client stores.

To read a region [read()], the client consults the region map and issues RDMA read(s) to the proper host(s); it then copies the result to the user-provided buffer. To write a region [write()], the client checks if the write falls outside the preallocated space for the region; if it does, it contacts the manager to extend the region and the region map; then, the library copies the user-provided buffer into RDMA-registered memory, consults the region map to determine the host(s) to contact, and issues RDMA-write(s) to the proper host(s).

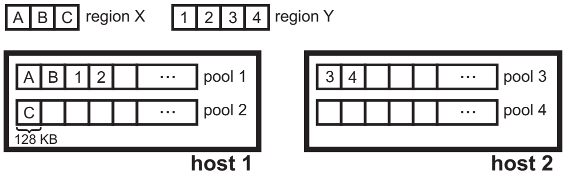


Figure 8: Four pools in two hosts. Region X stores its data in two pools of a host. Region Y is multi-hosted, spanning pools of two different hosts.

To prefetch data [ioctl for rprefetch()], the client consults the region map to determine where to read the data from, reads over RDMA, and places it in the file system cache. Similarly, to write back [msync()] a page, the client consults the region map, write-protects the page, writes the page over RDMA, and marks the page clean. To mark a page dirty [ioctl for rmarkdirty()], the client sets the dirty bit for the page. To clear a page from the cache [ioctl for rclearcache()], the client evicts it from the file system cache.

6.7 Pools

A *pool* is a chunk of physical memory, at one of the hosts, that is used to store a region or parts thereof (Figure 8). Pools are allocated by the daemon (§6.4) and are shared with local processes (using shared memory) and with remote processes (using RDMA). To share locally, the daemon allocates its pools using anonymous files [38], which are chunks of anonymous memory that can be memory mapped at many processes. More precisely, the daemon creates a pool using memfd_create; then, an application process can memory map the pool at the addresses that correspond to a region that the process needs. Regions need not be contiguous within a pool; however, to reduce the number of memory maps, the daemon allocates the region in large contiguous chunks.

To share its pools with remote hosts, the daemon RDMA-registers each pool so that it can be read and written over RDMA. RDMA provides access control through a key for each buffer that a host exports. Because a pool is a single buffer, this mechanisms is coarse-grained: it provides identical access to all data in the pool.

6.8 Finding region data

A *region map* tracks where a region is stored, by mapping offsets in a region to a host, a pool in that host, and an offset in that pool. The map has entries, each representing a fixed-length contiguous chunk of memory on one pool at one host. There are two aspects to this map: its granularity and how to represent hosts and pools.

Granularity. The granularity involves a trade-off between the size and the flexibility of the map. A small grain leads to a prohibitively large map; a large grain leads to internal fragmentation. A natural size might be the page size (4 KB), but that causes a significant 0.2% space overhead for the map (e.g., 2 GB for a 1 TB region). We chose the granularity to be 128 KB for an overhead of 64 KB for a 1 TB region.

Target representation. We want map entries to take at most 64 bits. We use 47 bits to represent a 64-bit address within a host, by dropping the lower 17 bits and aligning over 2^{17} =128 KB chunks, which coincides with the map granularity above. We use the remaining 17 bits as a global identifier that maps to a host and a pool in that host; this map is kept by the manager ("pool list" box in Figure 7) and cached by the user library.

6.9 Managing memory

There are two aspects to memory management: local and cluster allocation.

Local allocation. Each host has limited memory and one needs to decide how much to reserve for pools and regions. We make this determination locally at each host, where the daemon allocates and frees pools as needed.

Cluster allocation. When an application needs memory, one needs to decide which pool(s) to use; for multi-hosted regions, one needs to also decide which hosts will provide memory. We make this determination in a centralized fashion: the manager knows about all participating hosts, their pools, and the free space in each pool ("pool allocator" box in Figure 7). The manager receives requests to create new regions, with an initial space to preallocate for future region growth. It then decides from what pools to allocate the memory using some allocation policy. The current policy is as follows. For regions in a single host, the manager picks from the pools in that host; if the host does not have enough memory, it asks the daemon to create more pools; if the daemon is unable, the request to create or expand a region fails. For multi-hosted regions (§5.6), the manager picks memory from the hosts in a round-robin fashion, allocating ALLOCSIZE$\geq 2^{17}$ bytes at a time. Note that ALLOCSIZE becomes the maximum contiguous size that a client can transfer in one RDMA request. We pick ALLOCSIZE to be 2 MB—a value large enough to offset the initial fixed costs of an RDMA transfer (with a 40 Gbps network, the initial cost to transfer 2 MB is 0.3% of the total cost).

6.10 Allocating virtual addresses

Regions are assigned a fixed and unique virtual address (§5.7). Therefore, we must ensure that (a) different regions get assigned disjoint virtual addresses, even if they are created by different applications in different hosts, and (b) an application will not use a region's virtual addresses for other purposes. To ensure (a), we use centralization: region creation goes through the manager, who knows about all virtual addresses in use by regions. The manager assigns unique virtual addresses to each region ("vaddr allocator" box in Figure 7). To ensure (b), we reserve a range of virtual addresses for regions using the dynamic linker responsible for loading binaries. There are many ways to do that in Linux. First, we can specify an ET_EXEC object file type in the ELF binary and then create a program header with attributes p_vaddr and p_memsz, indicating the address and size of the virtual address to reserve [19]; this requires statically linking all libraries. Second, we can use a custom dynamic linker that avoids the virtual addresses reserved for regions; we do that by including in the ELF binary an INTERP program header with the path to the linker [16]. These approaches require modifying the application binary. A third approach, which requires no binary changes, is to modify the default dynamic linker, ld-linux.so.

Are there enough virtual addresses? Today, Intel processors use page tables with four levels, addressing 48 bits of addresses; one bit is used by the Linux kernel, leaving 47 bits for applications. If we reserve another bit for regions, that leaves 64 TB for each application and 64 TB for all regions. If that is not enough, Intel plans to support five-level page tables, which add 9 bits of virtual addressing [55]; reserving one bit for regions gives 32 PB for each application and 32 PB for all regions.

6.11 Security

We enforce access control using the file system, which assumes that the kernel is trusted. This provides reasonable security against damage from bugs and human errors, but an attacker of a host gets access to the regions in every host. Providing stronger security is future work.

6.12 Other modules

The kernel lib consists of two kernel modules: (1) RPC comm module implements RPC's to the manager, and (2) the RDMA comm modules establishes a reliable RDMA connection to remote hosts and implements one-sided RDMA read and write. The RPC handler module at the manager handles RPC requests from clients. The RDMA handler at the daemons registers the pools with RDMA, reports the RDMA key and pool address to the manager so that clients can later access the pool, and accepts reliable RDMA connections from remote daemons.

System	Description
rdma	RDMA read or write
nfs-tmpfs	NFS to a ramdisk
tmpfs	local ramdisk
rr	regions
rr+	regions with prefetching

Figure 9: Baselines (top) and systems under study (bottom).

6.13 User-level file interface

An earlier version of REGIONFS was implemented as a user-level file system [42] and a user-level page-fault handler [62]. We found that data operations were faster, because they could use RDMA's user-level interface. However, page fault handling was slower. As future research, it would be interesting to explore a hybrid design that provides both user-level and kernel interfaces to the *same* file system to get the best of both worlds.

7 Implementation

We implemented a prototype of regions for the Linux kernel v4.8 with 7700 lines of C/C++. Our current implementation differs from the design in a few significant ways: (1) we do not replicate the manager, (2) at each daemon, we have a fixed number of pools, hence a fixed amount of memory for regions, and (3) our VFS file system implements only the functionality needed to run our applications and benchmarks.

8 Evaluation

Our goal is to understand how well do regions perform, and how easy it is to use them in practice. To answer these questions, we use micro-benchmarks, examine code complexity, modify two applications to use regions, and measure their performance.

8.1 Testbed

Our testbed has 8 machines connected to a 100 Gbps RoCE switch. Each server has 128 GB RAM, a 800 GB SATA SSD, dual Intel Haswell-EP 2.4 GHz processors with a Mellanox ConnectX-4 NIC and Linux kernel 4.8.

8.2 Baselines

We compare the performance of regions against three baselines (Figure 9). RDMA offers a different interface to remote memory (RDMA verbs). Nfs-tmpfs and tmpfs provide a similar interface as regions (files), but without access to remote memory: nfs-tmpfs accesses files in a RAM disk of the NFS server, while tmpfs accesses files in a local RAM disk without network overheads, representing an upper bound on achievable performance.

We consider two variants of regions (rr and rr+) without and with performance enhancements that we describe in each experiment. In all experiments, we configure a region to be stored remotely from the benchmark or application operating on it, and so they access the region over the network rather than locally.

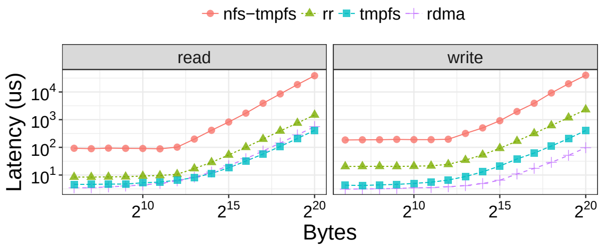


Figure 10: Latency for transferring data (no caching).

8.3 Performance of memory-mapped access

Setup. We study the time it takes for regions to read and write memory-mapped data. In an experiment, we memory map a file or region, and then sequentially read or write bytes. We choose an operation type (read or write), and operation size (number of bytes to read or write), and repeat the operation 100 times, measuring the latency of each operation. We compare regions against the baselines (nfs-tmpfs, tmpfs, rdma); RDMA does not support memory-mapping, so we instead read or write the data using one-sided RDMA verbs. We consider two variations of region. One variant (rr) performs raw operations without caching: we drop the cache after every operation, so that every operation must go over the network. The other variant (rr+) caches the most recently accessed page, so that consecutive operations on the same page access the cache, and writes to a page are buffered until the entire page is written. We also consider a variant of nfs-tmpfs that caches a page in the same way (nfs-tmpfs+).

Results. Figure 10 shows the results. For reads (left), we see that nfs-tmpfs and rr are flat from 64 bytes until 4K; this is because the file system operates at a page granularity, so it fetches an entire page even if the request needs fewer bytes. RDMA and tmpfs have the lowest latency, at 41% and 54% of rr's latency on 64 bytes, and 38% and 28% on 1MB. This is because rr suffers from overheads of page faults, 4KB-transfer granularity, and the file cache; tmpfs also incurs those overheads, but it compensates by avoiding the network latency. nfs-tmpfs is the worst due to higher network overheads. For writes (right), results are qualitatively similar; for rr and nfs-tmpfs, writes are slower than reads because the file system performs a read-modify-write operation, where it first reads the page before it writes it, requiring two network round trips. With RDMA, writes are faster than reads because RDMA writes complete as soon as they are posted on the PCIe bus at the remote host, whereas reads

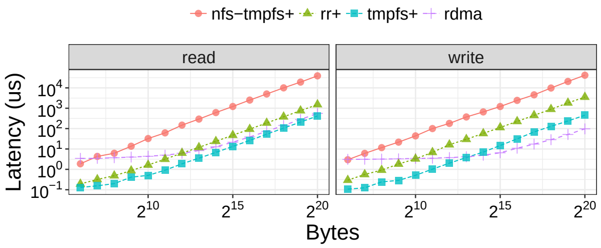


Figure 11: Latency for transferring data (caching).

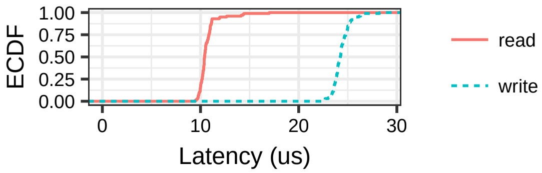


Figure 12: Distribution of latency for reading and writing 4 KB on a memory-mapped region using rr.

need to get data from memory.

These comparisons are unfair to file systems: unlike RDMA, they fetch full 4KB pages even for small requests, and cache them; doing so benefits applications that use the page later, but Figure 10 gives no credit for that. So, we now consider the effects of having a cache. Figure 11 shows the results with caching of the last page accessed. We see much improvement for small requests. Here, rr+ performs better than RDMA up to 4KB (reads) or 1KB (writes). We include RDMA in the graph for comparison, but it has no cache. Theoretically, an application can implement its own caching for RDMA, but doing so makes RDMA even more complex. In contrast, caching comes for free with a file system, without any application effort.

Figure 12 shows the cdf for reading or writing 4 KB of data using rr (no cache). We see a concentration from 10–11 us for reads, with the 95-percentile at 12 us; and a concentration from 23–26 us for writes, with the 95-percentile at 25.8 us. As we pointed out, writes are slower because the file system performs a read-modify-write operation (with memory-mapping, the system does not know that the application will eventually overwrite the entire page). This overhead is avoided by calling rmarkdirty (§5.8) prior to writing a page (not shown).

8.4 Performance of the file system

Setup. We run Sysbench, a standard file IO benchmark [59], to measure the performance of reading data from REGIONFS. We configure Sysbench with a single thread that reads from a 2GB file and reports throughput, average latency, and 95% latency. We study sequential and random reads of 16 KB chunks. We compare REGIONFS against nfs-tmpfs and local tmpfs.

System	Seq Tput (MB/s)	Seq LatAve (ms)	Seq Lat95 (ms)	Rnd Tput (MB/s)	Rnd LatAve (ms)	Rnd Lat95 (ms)
nfs-tmpfs	4871	0	0.01	4247	0	0.01
rr	5432	0	0.01	4821	0	0.01
tmpfs	6556	0	0	6048	0	0

Figure 13: Sysbench file IO benchmark results. Seq refers to sequential reads, Rnd to random reads.

Functionality	Description	regions loc	RDMA loc
Initialization	Code needed in every application	6	229
Producer-consumer	Simple message queue	29	103
Linked list	Traverse linked list	18	68
Hash table	Lookup operation of hash table	14	78
Access revocation	Remove access from a host	1	37

Figure 14: Equivalent functionality in regions and RDMA.

Results. Figure 13 shows the results. For throughput, tmpfs performs the best: 6.5 and 6.0 GB/s for sequential and random reads, respectively, while REGIONFS is within 83% and 80%—reasonably close to the in-memory performance of tmpfs, despite going to the network. Nfs-tmpfs is the worst at 4.9 and 4.2 GB/s. For latency, the resolution of the benchmark is 0.01ms, which is too large to reflect the difference between the different systems. (Please refer to the previous experiments, where we report other latency numbers.)

8.5 Code complexity

Setup. To study code complexity, we implement functionality that is commonly used in remote memory applications, and compare the number of lines of code (LOC) to implement them using REGIONFS and RDMA verbs.

Results. Figure 14 shows the complexity results. We see that region code has much fewer lines of code – 4.2 times on average, excluding initialization and revocation. For initialization, region requires just opening and memory mapping a file, while RDMA requires initializing contexts, memory registration, establishing connections, creating, transitioning and initializing queue pairs, key exchange, and more. For the other functionality, regions are similarly simpler, requiring just memory or file operations, while RDMA code must manually submit requests to queue pairs, monitor for completions, etc. In addition, RDMA verbs require explicit management of a partitioned global address space, which translates to more work at the application level. This complexity makes it hard for new developers to even get started on RDMA.

Next, we further study complexity, by using regions to adapt two applications to use remote memory.

8.6 Application: R

R is a statistical processing system for data in memory. Using regions, we adapt R to use remote memory. R

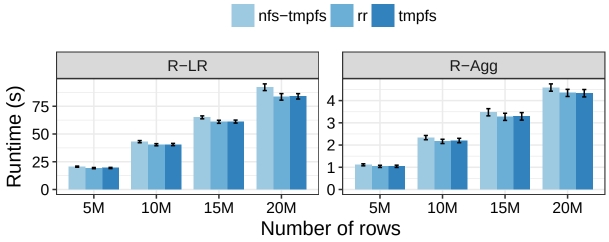


Figure 15: R application. Bars show total runtime (smaller is better). Error bars show std deviation.

has a large number of packages to extend its core functionality, including a package *ff* that extends R's memory capacity using memory-mapped files. ff provides objects that are each stored in a file; to limit memory consumption, small parts of the file called *sections* get memory mapped and unmapped as needed, with at most one section per file mapped at a time. We set up ff to use the files in REGIONFS, and to use sections that are 128KB wide.

Setup. In each experiment, we choose a workload for R and an input size. We use R to process the workload with an input of the given size, and we measure the time it takes. The workload mimics a data analyst that has a large data set and uses R to analyze parts of it. The data set is a large matrix stored in a host different from the one running R, representing data generated elsewhere in the network that does not fit in R's memory. The matrix is stored as several ff objects, one per column, with 200 columns and a number of rows that varies from 5 to 20 million. We consider two workloads:

- *R-Agg.* Compute an aggregation (mean) over ten columns of the matrix. This workload represents an extreme in terms of the ratio of computation to memory accesses: it almost entirely performs memory accesses, by reading data and only computing a sum.
- *R-LR.* Compute a linear regression over ten columns of the matrix. The algorithm accesses the rows of the matrix several times, but performs significant more computation than R-Agg, representing a balance between memory accesses and compute.

We consider three systems: rr, tmpfs, and nfs-tmpfs.

Results. Figure 15 (right) shows the R-Agg workload. Regions approach tmpfs within 1%, despite having to read the input from a remote host. In comparison, nfs-tmpfs is within 6% of tmpfs.

For the R-LR workload (Figure 15 left), we see a similar trend but with a larger running time incurred by linear regression. Regions are again within 1% of tmpfs, while nfs-tmpfs is within 9% of tmpfs.

While the performances of tmpfs and nfs-tmpfs are similar to rr, tmpfs and nfs-tmpfs do not permit R to run with a large memory because their capacity is limited by the available memory locally or in the nfs server. By contrast, regions can be multihosted (§5.4) to aggregate the memory of many machines.

We also ran R and placed the ff-generated files in an SSD rather than in REGIONFS, as a way to obtain more space. The running time of R increased by 2.5× (for R-agg) and 2.7× (for R-LR) relative to rr.

As for code complexity, we made no changes to R's ff package; we just set it up to use files in REGIONFS.

8.7 Applications: Metis

Metis is an in-memory map-reduce processing framework. Metis reads its initial input from a file, and launches many threads to run map-reduce. In map-reduce, the data is partitioned across a set of mappers, each producing an output; the outputs of all mappers are grouped based on a key, and the groups are partitioned across the reducers; each reducer produces some output, and all outputs are aggregated in the end. Metis runs on a single host, using work queues to distribute map and reduce tasks among many threads.

We modify Metis to run across many hosts, using regions to share its input and output. More specifically, the modified Metis does three things: (1) reads the initial input from remote hosts using regions, representing data produced by another computation, such as a previous map-reduce job[1], (2) runs threads across many hosts, with each host writing the output to a region to make it available to the other hosts, and (3) collects the regions with the results from all the hosts and aggregates the output. In effect, we produce a distributed version of Metis, while retaining its in-memory processing.

Setup. In each experiment, we run a map-reduce job to produce a histogram. In this job, each mapper processes a partition of the input and produces a partial histogram; the reducers then aggregate the bins, each reducer responsible for a disjoint set of bins; a final stage collects the bins from the reducers.

We consider two systems: the original Metis and our distributed version. We vary the number of threads in each system; for the distributed version, we vary the number of hosts. We measure the time to run the map-reduce computation. The input is a 2.6 GB image file, and the output produces 403 bins. Metis has an option called *prefault* to initially preload all input to memory.

Results. Figure 16 (left) shows the results for a single host as we increase the number of threads. We see that rr is faster because it reads the input from remote memory, while Metis reads from an SSD. For one host and one thread, rr is 2.0× faster than Metis; for 4 threads, it is 3.5× faster. By increasing the number of threads to 4,

[1] The motivation is that many map-reduce applications run a chain of map-reduce jobs, each consuming the output of the previous job.

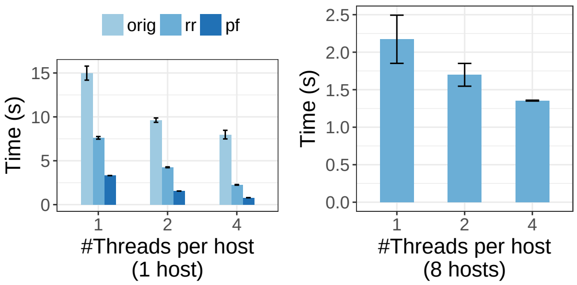


Figure 16: Metis application performance. On the left, there is a single host; *rr* refers to our distributed version of Metis, while *orig* and *pf* refer to the original Metis, where *pf* enables Metis's prefault option. On the right, we show our distributed version of Metis on 8 hosts (there are no bars for the original Metis since it runs only on 1 host). The y-axis is running time (smaller is better). Error bars show std deviation.

Metis and rr improve by 1.91× and 3.4×. If we exclude reading the input (with prefault), performance improves by 2.3–2.9× compared to rr.

Figure 16 (right) shows the results for our distributed version of Metis running on eight hosts and 1–4 threads per host. We see that performance improves compared to running with one host. Using 8 hosts, rr is 3.5, 2.5, 1.7× faster for 1, 2, 4 threads than rr using a single host with the same number of threads. The improvement comes from the hosts running the computation in parallel. The scalability is not linear because the running time is only a few seconds and so the overhead of synchronizing the hosts between phases is relatively high.

As for code complexity, the modifications to Metis consist of 82 lines. This is small, as the changes amount to changing a centralized system into a distributed one.

9 Conclusion

In this paper, we applied the Unix idea that "everything is a file" to remote memory, obtaining an abstraction in which a process exports parts of its memory as a file that remote processes can access. We studied the design behind this abstraction, described a prototype that achieves reasonable performance, and showed that applications can easily benefit from it.

References

[1] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *ACM Symposium on Cloud Computing*, pages 121–127, Sept. 2017.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[3] Apache Hadoop. https://hadoop.apache.org/, July 2008.

[4] K. Asanovic and D. Patterson. FireBox: A hardware building block for 2020 warehouse-scale computers. In *Keynote USENIX Conference on File and Storage Technologies*, Feb. 2014.

[5] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems*, 16(1):1–40, Feb. 1998.

[6] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *International Conference on Management of Data*, pages 1463–1475, May 2015.

[7] T. Bedeir. Building an RDMA-capable application with IB verbs. RDMA read and write with IB verbs. https://thegeekinthecorner.wordpress.com/2013/02/02/rdma-tutorial-pdfs, Aug. 2010.

[8] C. G. Bell and I. Nassi. Revisiting scalable coherent shared memory. *IEEE Computer*, 51(1):40–49, Jan. 2018.

[9] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Mar. 1990.

[10] N. Carriero and D. Gelernter. The S/Net's Linda kernel (extended abstract). In *ACM Symposium on Operating Systems Principles*, page 160, Dec. 1985.

[11] Ceph file system. https://ceph.com/ceph-storage/file-system.

[12] D. Comer and J. Griffioen. A new design for distributed systems: The remote memory model. In *Usenix Summer 1990 Technical Conference*, pages 127–136, June 1990.

[13] Direct access file system (DAFS) protocol. http://www.dafscollaborative.org.

[14] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *Symposium on Networked Systems Design and Implementation*, pages 401–414, Apr. 2014.

[15] A. Dragojević, D. Narayanan, E. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles*, pages 54–70, Oct. 2015.

[16] U. Drepper. How to write shared libraries. https://www.akkadia.org/drepper/dsohowto.pdf.

[17] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *International Parallel Processing Symposium*, pages 153–159, Apr. 1999.

[18] Dynamically connected transport. https://www.openfabrics.org/images/eventpresos/workshops2014/DevWorkshop/presos/Monday/pdf/05_DC_Verbs.pdf, 2014.

[19] Executable and linkable format. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

[20] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic. Beyond processor-centric operating systems. In *Workshop on Hot Topics in Operating Systems*, pages 17–17, May 2015.

[21] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *ACM Symposium on Operating Systems Principles*, pages 201–212, Dec. 1995.

[22] M. D. Flouris and E. P. Markatos. The network RamDisk: Using remote memory on heterogeneous nows. *Cluster Computing*, 2(4):281–293, Oct. 1999.

[23] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *Symposium on Operating Systems Design and Implementation*, pages 249–264, Oct. 2016.

[24] Gen-Z draft core specification—december 2016. http://genzconsortium.org/draft-core-specification-december-2016.

[25] GlusterFS documentation. http://docs.gluster.org/en/latest.

[26] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *Symposium on Networked Systems Design and Implementation*, pages 649–667, Mar. 2017.

[27] Y. C. Hu, W. Yu, A. Cox, D. Wallach, and W. Zwaenepoel. Runtime support for distributed sharing in safe languages. *ACM Transactions on Computer Systems*, 21(1):1–35, Feb. 2003.

[28] Intel Omni-Path. http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html.

[29] iWARP. https://en.wikipedia.org/wiki/IWARP.

[30] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 295–306, Aug. 2014.

[31] S. Koussih, A. Acharya, and S. Setia. Dodo: A user-level system for exploiting idle memory in workstation clusters. In *IEEE International Symposium on High Performance Distributed Computing*, pages 301–308, Aug. 1999.

[32] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[33] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[34] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over InfiniBand: An approach using a high performance network block device. In *IEEE International Conference on Cluster Computing*, pages 1–10, Sept. 2005.

[35] libibverbs. http://www.rdmamojo.com/2012/05/18/libibverbs.

[36] E.-J. Lim, S.-Y. Ahn, Y.-H. Kim, G.-I. Cha, and W. Choi. Design of cache backend using remote memory for network file system. In *International Conference on High Performance Computing and Simulation*, pages 864–869, July 2017.

[37] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *International Symposium on Computer Architecture*, pages 267–278, June 2009.

[38] Linux man page for memfd_create (2).

[39] X. Lu, N. S. Islam, M. Wasi-ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-performance design of hadoop RPC with RDMA over infiniband. In *IEEE International Conference on Parallel Processing*, pages 641–650, Oct. 2013.

[40] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *USENIX Annual Technical Conference*, pages 773–785, July 2017.

[41] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-2010-020, MIT, May 2010.

[42] D. Mazières. A toolkit for user-level file systems. In *USENIX Annual Technical Conference*, pages 261–274, June 2001.

[43] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, June 2013.

[44] L. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. Technical Report CMU-CS-94-213, Carnegie Mellon University, Nov. 1994.

[45] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *Symposium on Networked Systems Design and Implementation*, pages 17–33, Mar. 2017.

[46] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference*, pages 291–305, July 2015.

[47] OpenCAPI consortium. http://opencapi.org.

[48] The R project for statistical computing. https://www.r-project.org.

[49] RDMA over converged ethernet. https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet.

[50] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *Proceedings of the VLDB Endowment*, 9(4):228–239, Dec. 2015.

[51] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Oct. 1996.

[52] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[53] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, Oct. 1994.

[54] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed shared persistent memory. In *ACM Symposium on Cloud Computing*, pages 323–337, Sept. 2017.

[55] K. A. Shutemov. [patchv4 9/9] x86/mm: Allow to have userspace mappings above 47-bits. http://www.spinics.net/lists/linux-mm/msg125594.html.

[56] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: software coherent shared memory on a clustered remote-write network. In *ACM Symposium on Operating Systems Principles*, pages 170–183, Oct. 1997.

[57] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas. Crail: A high-performance I/O architecture for distributed data processing. *IEEE Data Engineering Bulletin*, 40(1):38–49, Mar. 2017.

[58] M. M. Swift. Towards O(1) memory. In *Workshop on Hot Topics in Operating Systems*, pages 7–11, May 2017.

[59] sysbench. https://github.com/akopytov/sysbench.

[60] T. Talpey and C. Juszczak. Network file system (NFS) remote direct memory access (RDMA) problem statement. RFC 5532, RFC Editor, May 2009.

[61] S.-Y. Tsai and Y. Zhang. LITE kernel RDMA support for datacenter applications. In *ACM Symposium on Operating Systems Principles*, pages 306–324, Oct. 2017.

[62] Userfaultfd. https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt.

[63] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems, Nov. 1994.

[64] E. Zamanian, C. Binnig, T. Kraska, and T. Harris. The end of a myth: Distributed transactions can scale. *Proceedings of the VLDB Endowment*, 10(6):685–696, Feb. 2017.

Understanding Ephemeral Storage for Serverless Analytics

Ana Klimovic[1], Yawen Wang[1], Christos Kozyrakis[1],
Patrick Stuedi[2], Jonas Pfefferle[2], and Animesh Trivedi[2]

[1]Stanford University
[2]IBM Research

Abstract

Serverless computing frameworks allow users to launch thousands of concurrent tasks with high elasticity and fine-grain resource billing without explicitly managing computing resources. While already successful for IoT and web microservices, there is increasing interest in leveraging serverless computing to run data-intensive jobs, such as interactive analytics. A key challenge in running analytics workloads on serverless platforms is enabling tasks in different execution stages to efficiently communicate data between each other via a shared data store. In this paper, we explore the suitability of different cloud storage services (e.g., object stores and distributed caches) as remote storage for serverless analytics. Our analysis leads to key insights to guide the design of an ephemeral cloud storage system, including the performance and cost efficiency of Flash storage for serverless application requirements and the need for a pay-what-you-use storage service that can support the high throughput demands of highly parallel applications.

1 Introduction

Serverless computing is an increasingly popular execution model in the cloud. With services such as AWS Lambda, Google Cloud Functions, and Azure Functions, users write applications as collections of stateless functions which they deploy directly to a serverless framework instead of running tasks on traditional virtual machines with pre-allocated resources [8, 14, 19, 2]. The cloud provider schedules user tasks onto physical resources with the promise of automatically scaling according to application demands and charging users only for the fine-grain resources their tasks consume.

While already popular for web microservices and IoT applications, the elasticity and fine-grain billing advantages of serverless computing are also appealing for a broader range of applications, including interactive data analytics. Several frameworks are being developed which leverage serverless computing to exploit high degrees of parallelism in analytics workloads and achieve near real-time performance [13, 17, 10].

A key challenge in running analytics workloads on serverless computing platforms is efficiently sharing data between tasks. In contrast to simple event-driven applications that consist of a single task executed in response to an event trigger, analytics workloads typically consist of multiple stages and require intermediate results to be shared between stages of tasks. In traditional analytics frameworks (e.g., Spark, Hadoop), tasks buffer intermediate data in local storage and exchange data between tasks directly over the network [25, 24]. In contrast, serverless computing frameworks achieve high elasticity and scalability by requiring tasks to be stateless [15]. In other words, a task's local file system and child processes are limited to the lifetime of the task itself. Furthermore, since serverless platforms do not expose control over task scheduling and placement, direct communication between tasks is difficult. Thus, the natural approach for inter-task communication is to store intermediate data in a common, remote storage service. We refer to data exchanged between tasks as *ephemeral data*.

There are several storage options for data sharing in serverless analytics jobs, each providing different cost, performance and scalability trade-offs. Managed object storage services like S3 offer pay-what-you-use capacity and bandwidth for storage resources managed by the provider [7]. Although primarily intended for long term data storage, they can also be used for ephemeral data. In-memory key-value stores like Redis and Memcached offer high performance, at the high cost of DRAM [21, 4]. They also require users to manage their own storage VMs. It is not clear whether existing storage options meet the demands of serverless analytics or how we can design a storage system to rule them all.

In this paper, we characterize the I/O requirements for data sharing in three different serverless applications including MapReduce sort, distributed software compilation, and video processing. Using AWS Lambda as our serverless platform, we analyze application performance using three different types of storage systems. We consider a disk-based, managed object storage service (Amazon S3), an in-memory key value store (ElastiCache Redis), and a Flash-based distributed storage system (Apache Crail with a ReFlex Flash backend [1, 23, 18]). Our analysis leads to key insights for the design of distributed ephemeral storage, such as the use of Flash to cost-efficiently support the throughput, latency and capacity requirements of most applications and the need for a storage service that scales to meet the demands of applications with abundant parallelism. We conclude with a discussion of remaining challenges such as resource auto-scaling and QoS-aware data placement.

2 Serverless Analytics I/O Properties

We study three different serverless analytics applications and characterize their throughput and capacity requirements, data access frequency and I/O size. We use AWS Lambda as our serverless platform and configure lambdas with the maximum supported memory (3 GB) [8]. Figure 1 plots each job's cumulative storage bandwidth usage over time. Figure 2 shows the I/O size distribution.

Parallel software build: We use a framework called gg to automatically synthesize the dependency tree of a software build system and coordinate lambda invocations for distributed compilation [12, 3]. Each lambda fetches its dependencies from ephemeral storage, computes (i.e., compiles, archives or links depending on the stage), and writes an output file. Compilation stage lambdas read source files which are generally up to 10s of KBs. While 55% of files are read only once (by a single lambda), others are read hundreds of times (by many lambdas in parallel), such as glibc library files. Lambdas which archive or link read objects up to 10s of MBs in size. We use gg to compile `cmake` which has 850 MB of ephemeral data.

MapReduce Sort: We implement a MapReduce style sort application on AWS Lambda, similar to PyWren [17]. Map lambdas fetch input files from long-term storage (S3) and write intermediate files to ephemeral storage. Reduce lambdas merge and sort intermediate data read from ephemeral storage and write output files to S3. Sorting is I/O-intensive. For example, we measure up to 7.5 cumulative GB/s when sorting 100 GB with 500 lambdas. Each intermediate file is written and read only once and its size is directly proportional to the dataset size and inversely related to the number of workers.

Video analytics: We use Thousand Island Scanner (THIS) to run distributed video processing on lambdas [20]. The input is an encoded video that is divided into batches and uploaded to ephemeral storage. First stage lambdas read a batch of encoded video frames from ephemeral storage and write back decoded video frames. Each lambda then launches a second stage lambda which reads a set of decoded frames from ephemeral storage, computes a MXNET deep learning classification algorithm and outputs a classification result. We use a video consisting of 6.2K 1080p frames and tune the batch size to optimize runtime (62 lambdas in the decode stage and 310 lambdas for classification). The total ephemeral storage capacity is 6 GB.

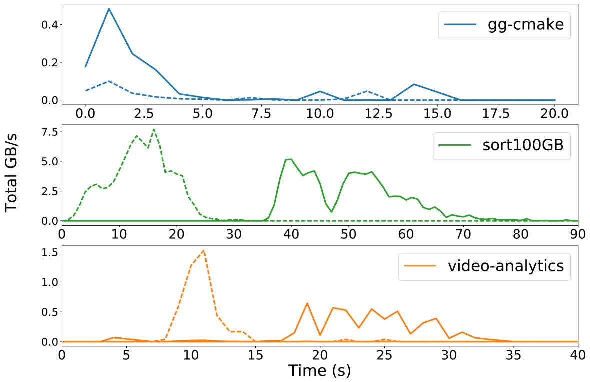


Figure 1: Cumulative throughput over time.

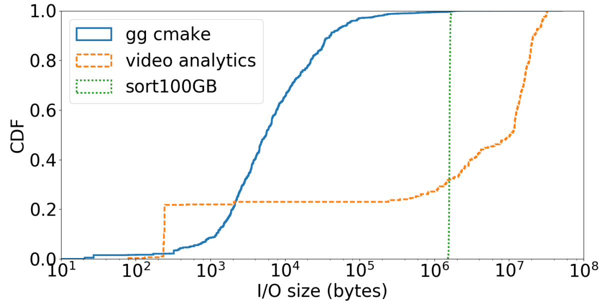


Figure 2: I/Os range from 100s of bytes to 100s of MBs.

3 Remote Storage for Data Sharing

We consider three different categories of storage systems for ephemeral data sharing in serverless analytics: fully managed cloud storage (e.g., S3), in-memory key-value storage (e.g., Redis), and distributed Flash storage (e.g., Crail-ReFlex). We focus on ephemeral storage as the original input and final output data of analytics jobs typically has long-term availability requirements that are well served by various existing cloud storage systems.

Simple Storage Service (S3): Amazon S3 is a fully managed object storage system that achieves high availability and scalability by replicating data across multiple nodes with eventual consistency [9]. Users pay only for the storage capacity and bandwidth they use, without ex-

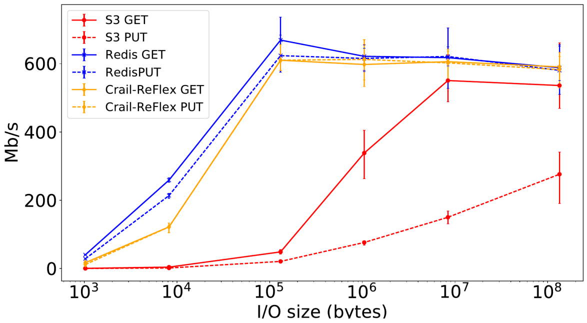


Figure 3: Peak storage throughput per lambda

	Read	Write	Metadata lookup
S3	12.1 ms	25.8 ms	–
Redis	230 μs	232 μs	–
Crail-ReFlex	283 μs	386 μs	185 μs

Table 1: Average unloaded latency for 1KB requests.

plicitly managing storage resources. S3 has significant overhead, particularly for small requests. As shown in Table 1, it takes on average over 25 ms to write 1KB. For requests smaller than 100 KB, Figure 3 shows that a single lambda achieves less than 5 MB/s (40 Mb/s) throughput. For requests 10 MB or larger, throughput goes up to 70 MB/s. With up to 2500 concurrent lambda clients, S3 scales to 80 GB/s with each client achieving approximately 30 MB/s (not shown in the figure).

Elasticache Redis: DRAM is a viable storage media for ephemeral data, which is short-lived. We use ElastiCache Redis with cluster mode enabled as an example in-memory key-value store [21, 6]. Table 1 shows that Redis latency is ∼240 μs, two orders of magnitude lower than S3 latency. We find that AWS Lambda infrastructure introduces some overhead as the same `c4.8xlarge` Redis cluster has ∼ 115μs lower round-trip latency from a `r4.8xlarge` EC2 client (10 GbE). We also confirm the 640 Mb/s peak per-lambda throughput in Figure 3 is an AWS Lambda limitation; the EC2 client achieves up to 5 Gb/s for the same, single TCP connection test. Since we occasionally observe lambda throughput burst above 640 Mb/s, we suspect AWS throttles a 1 Gb/s link.

Crail-ReFlex: Finally, we consider a Flash-based distributed storage system as Flash offers a medium ground between disk and DRAM for both performance and cost. In particular, NVM Express (NVMe) Flash devices are becoming increasingly popular in the cloud, offering high performance and capacity per dollar [5]. We choose to use the Apache Crail distributed storage system as it is designed for high performance access to data with low durability requirements, which matches the properties of ephemeral data. While Crail is originally designed for RDMA networks which are not available on AWS, its modular architecture supports pluggable storage tiers. We implement a NVMe Flash storage tier for Crail based on ReFlex, an open-source software system for low-latency, high throughput access to Flash over commodity networks [18]. We deploy Crail-ReFlex on `i3` EC2 nodes. Table 1 shows that from a lambda client, remote access to Flash (using Crail-ReFlex) has similar read latency as remote access to DRAM (using Redis). However, while Redis uses a simple hash to assign keys to storage servers, Crail relies on metadata servers to route client requests and manage data placement across nodes for more control over load balancing and quality of service. Thus, Crail requires an extra round-trip for metadata lookup which takes 185 μs.

4 Serverless Analytics Storage Analysis

We compare three different storage systems for ephemeral data sharing in serverless analytics and discuss how application latency sensitivity, parallelism, and I/O intensity impact ephemeral storage requirements.

Latency-sensitive jobs: We find that jobs in which lambdas mostly issue fine-grain I/O operations are latency-sensitive. Out of the applications we study, only gg shows some sensitivity to storage latency since the majority of files accessed are under 100 KB. Figure 4 shows the runtime for a parallel build of `cmake` as a function of the number of concurrent lambdas (gg allows users to set the maximum lambda concurrency, similar to `-j` in `make`). The job benefits from the lower latency of Redis storage compared to S3 with up to 100 concurrent lambdas. The runtime with S3 and Redis converges as we increase concurrency because the job eventually becomes compute-bound on AWS Lambda.

Jobs with limited parallelism: While serverless platforms allow users to exploit high application parallelism by launching many concurrent lambdas, individual lambdas are wimpy. Hence, we find that jobs with inherently limited parallelism (e.g., due to dependencies between lambdas) are likely to experience lambda resource bottlenecks (e.g., memory, compute and/or network bandwidth limits) rather than storage bottlenecks. This is the case for gg . The first stage of the software build process has high parallelism as each file can be pre-processed, compiled and assembled independently. However, subsequent lambdas which archive and link files depend on the outputs of earlier stages. Figure 5 plots the per-lambda read, compute and write times when using gg to compile `cmake` with up to 650 concurrent lambdas (650 is the highest degree of parallelism in the job's dependency graph). Using Redis (Figure 5b) compared to S3 (Figure 5a) reduces the average time that lambdas spend on I/O from 51% to 11%. However, the job takes approx-

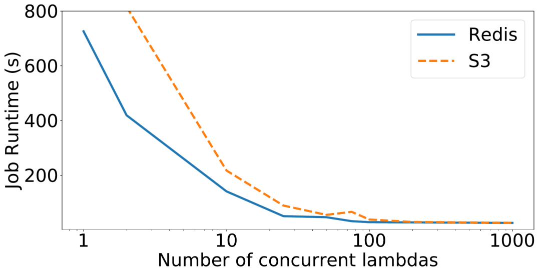
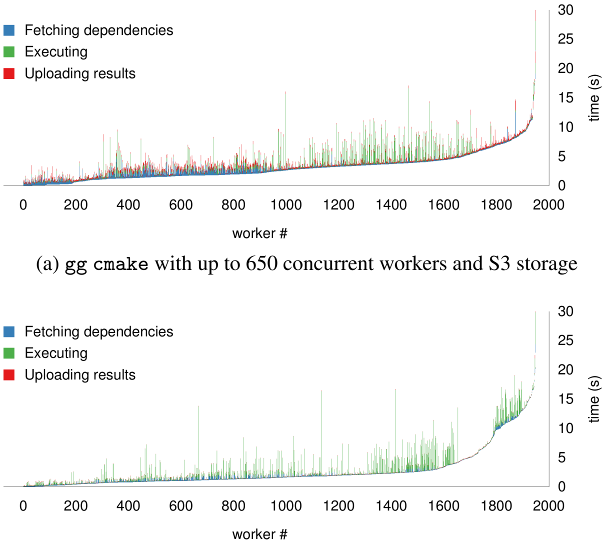


Figure 4: Distributed compilation of `cmake` is latency-sensitive at low concurrency and becomes compute-bound when run with ∼100 or more concurrent lambdas.

imately 30 seconds to complete, regardless of the storage system. This is because optimizing I/O does not affect the lambdas with particularly high compute latency which become the bottleneck.

Throughput-intensive jobs: MapReduce sort is an I/O-intensive application with abundant parallelism. Figure 6 shows the average time each lambda spends on I/O and compute to sort a 100 GB dataset [16]. We use S3 for input/output files and compare performance with S3, Redis (12 `cache.r4.2xlarge` nodes), Crail-ReFlex (12 `i3.2xlarge` nodes) as ephemeral storage. Storing ephemeral data in remote DRAM (Redis) or remote Flash (Crail-ReFlex) gives similar end-to-end performance, since we provision sufficient bandwidth in the storage clusters and the bottleneck becomes lambda CPU usage. Performance scales linearly as we increase the number of lambdas. S3 achieves lower throughput than Redis and Crail-ReFlex with 250 lambdas, leading to higher execution time. However, S3 outperforms a *single* node Redis or Crail-ReFlex cluster since a single node's network link becomes a bottleneck (not shown in the figure). Using S3 for ephemeral data shuffling with more than 250 lambdas in the 100 GB sort job results in I/O rate limit errors, preventing the job from completing.

Video analytics is another application with abundant parallelism. Figure 7 shows the average time lambdas in each stage spend reading, computing, and writing data. Reading and writing ephemeral data to/from S3 increases execution time compared to Redis and Crail-ReFlex. Stage 2 read time is higher with Crail-ReFlex than Redis due to read-write interference on Flash. Some lambdas in the first stage complete and launch second stage lambdas sooner than others. Thus read I/Os for some second stage lambdas interfere with the write requests from first stage lambdas that are still running. This interference can be problematic on Flash due to asymmetric read-write latency [18]. However, this does not noticeably affect overall performance as stage 2 lambdas are compute-bound. Stage 2 has low write time as its output (a list of objects detected in the video) is small.



(a) gg `cmake` with up to 650 concurrent workers and S3 storage

(b) gg `cmake` with up to 650 concurrent workers and Redis storage

Figure 5: Redis reduces I/O time compared to S3, but compute is the bottleneck. Based on Figure 6 from [12].

5 Discussion

Our analysis leads to several insights for the design of ephemeral storage for serverless analytics. We summarize the properties an ephemeral storage system should provide to address the needs of serverless analytics applications, make recommendations for the choice of storage media, and outline areas for future work.

Desired ephemeral storage properties: To meet the I/O demands of serverless applications, which can consist of thousands of lambdas in one execution stage and only a few lambdas in another, the storage system should have *high elasticity*. The system should also support *high IOPS* and *high throughput*. Since the granularity of data access varies widely (Figure 2), storing both small and large objects should be cost and performance efficient. To relieve users from the difficulty of managing storage clusters, the storage service should *auto-scale resources* based on load and charge users for the bandwidth and capacity used. This effectively extends the serverless abstraction to storage. Finally, the storage system can leverage the unique characteristics of ephemeral data. Namely, ephemeral data is short-lived and can easily be re-generated by re-running a job's tasks. Thus, unlike traditional long-term storage, an ephemeral storage system can provide low data durability guarantees. Furthermore, since the majority of ephemeral data is written and read only once (e.g., a mapper writes intermediate results for a particular reducer), the storage system can optimize capacity usage with an API that allows users to hint when data should be deleted right after it is read.

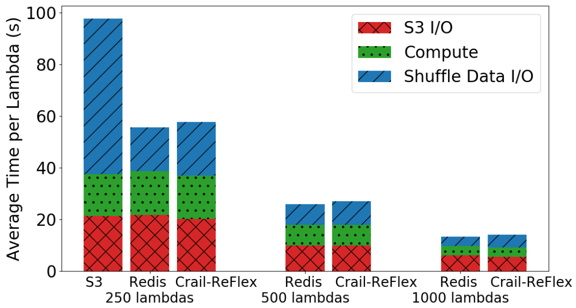


Figure 6: Average time per lambda for 100GB sort. S3 gives I/O rate limit errors with over 250 lambdas.

Choice of storage media: A storage system can support arbitrarily high throughput by scaling resources up and/or out. The more interesting question is which storage technology allows the system to cost effectively satisfy application throughput, latency and capacity requirements. Figure 1 plots cumulative GB/s over time for gg cmake, sort, and video analytics which have 0.85, 100, and 6 GB ephemeral datasets, respectively. Considering typical throughput-capacity ratios for each storage technology (DRAM: $\frac{20\text{GB/s}}{64\text{GB}} = 0.3$, Flash: $\frac{3.2\text{GB/s}}{500\text{GB}} = 0.006$, HDD: $\frac{0.7\text{GB/s}}{6\text{TB}} = 0.0001$), we conclude that the sort application is best suited for Flash-based ephemeral storage. The throughput-capacity ratios of the gg-cmake and video analytics jobs fall into the DRAM regime. However we observed that using Flash gives similar end-to-end performance for these applications at lower cost per GB, as lambda CPU is the bottleneck. I/O intensive applications with concurrent ephemeral read and write I/Os are likely to prefer DRAM storage as Flash tail read latency increases significantly with concurrent writes [18].

Future research directions: The newfound elasticity and fine resource granularity of serverless computing platforms motivates many systems research directions. Serverless computing places the burden of resource management on the cloud provider, which typically has no upfront knowledge of user workload characteristics. Hence, building systems that dynamically and autonomously rightsize cluster resources to meet elastic application demands is critical. The challenge involves provisioning resources across multiple dimensions (e.g., compute resources, network bandwidth, memory and storage capacity) in a fine-grain manner to find low cost allocations that satisfy application performance requirements. With multiple tenants sharing serverless computing infrastructure to run jobs with high fan-out, another challenge is providing predictable performance. Interference often leads to high variability, yet a job's runtime often depends on the slowest lambda [11]. Developing *fine-grain* isolation mechanisms and QoS-aware resource sharing policies is an important avenue to explore.

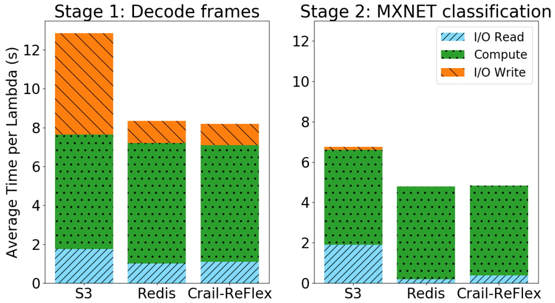


Figure 7: Video analytics I/O vs. compute breakdown, storing ephemeral data in S3, Redis and Crail-ReFlex.

6 Related Work

Fouladi et al. leverage serverless computing for distributed video processing and overcome the challenge of lambda communication by implementing the mu software framework to orchestrate lambda invocations with a long lived coordinator that is aware of each worker's state and execution flow [13]. While their system uses S3 to store ephemeral data, we study the suitability of three different types of storage systems for ephemeral data storage. Jonas et al. implement PyWren to analyze the applicability of serverless computing for generic workloads, including MapReduce jobs, and find storage to be a bottleneck [17]. We build upon their work, provide a more thorough analysis of ephemeral storage requirements for analytics jobs, and draw insights to guide the design of future systems. Singhvi et al. show that current serverless infrastructure does not support network intensive functions like packet processing [22]. Among their recommendations for future platforms, they also identify the need for a scalable remote storage service.

7 Conclusion

To support data-intensive analytics on serverless platforms, our analysis motivates the design of an ephemeral storage service that supports automatic and fine-grain allocation of storage capacity and throughput. For the three different applications we studied, throughput is more important than latency and Flash storage provides a good balance for performance and cost.

Acknowledgements

We thank the ATC reviewers for their feedback. We thank Sadjad Fouladi and Qian Li for the insightful technical discussions. This work is supported by the Stanford Platform Lab, Samsung and Huawei. Ana Klimovic is supported by a Stanford Graduate Fellowship.

References

[1] Apache crail (incubating). `http://crail.incubator.apache.org/`, 2018.

[2] Apache openwhisk (incubating). `https://openwhisk.apache.org/`, 2018.

[3] gg project. `https://github.com/stanfordsnr/gg`, 2018.

[4] Memcached – a distributed memory object caching system. `https://memcached.org/`, 2018.

[5] AMAZON. Amazon EC2 I3 instances, next-generation storage optimized high I/O instances. `https://aws.amazon.com/about-aws/whats-new/2017/02/now-available-amazon-ec2-i3-instances-next-generation-storage-optimized-high-i-o-instances`, 2017.

[6] AMAZON. Amazon ElastiCache. `https://aws.amazon.com/elasticache/`, 2018.

[7] AMAZON. Amazon simple storage service. `https://aws.amazon.com/s3`, 2018.

[8] AMAZON. AWS lambda. `https://aws.amazon.com/lambda`, 2018.

[9] AMAZON. Introduction to Amazon S3. `https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html`, 2018.

[10] DATABRICKS. Databricks serverless: Next generation resource management for Apache Spark. `https://databricks.com/blog/2017/06/07/databricks-serverless-next-generation-resource-management-for-apache-spark.html`, 2017.

[11] DEAN, J., AND BARROSO, L. A. The tail at scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80.

[12] FOULADI, S., ITER, D., CHATTERJEE, S., KOZYRAKIS, C., ZAHARIA, M., AND WINSTEIN, K. A thunk to remember: make -j1000 (and other jobs) on functions-as-a-service infrastructure (preprint). `http://stanford.edu/~sadjad/gg-paper.pdf`.

[13] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation* (2017), NSDI'17, pp. 363–376.

[14] GOOGLE. Cloud functions. `https://cloud.google.com/functions`, 2018.

[15] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[16] HIGGS, E. Terasort benchmark for spark. `https://github.com/ehiggs/spark-terasort`, 2018.

[17] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), SOCC'17, pp. 445–451.

[18] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. Reflex: Remote flash == local flash. In *Proc. of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ASPLOS '17, pp. 345–359.

[19] MICROSOFT. Azure functions. `https://azure.microsoft.com/en-us/services/functions`, 2018.

[20] QIAN LI, JAMES HONG, D. D. Thousand island scanner (THIS): Scaling video analysis on AWS lambda. `https://github.com/qianl15/this`, 2018.

[21] REDIS LABS. Redis. `https://redis.io`, 2018.

[22] SINGHVI, A., BANERJEE, S., HARCHOL, Y., AKELLA, A., PEEK, M., AND RYDIN, P. Granular computing and network intensive applications: Friends or foes? In *Proc. of the 16th ACM Workshop on Hot Topics in Networks* (2017), HotNets-XVI, pp. 157–163.

[23] STUEDI, P., TRIVEDI, A., PFEFFERLE, J., STOICA, R., METZLER, B., IOANNOU, N., AND KOLTSIDAS, I. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Engineering Bulletin 40*, 1 (2017), 38–49.

[24] WHITE, T. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[25] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (2010), HotCloud'10, pp. 10–10.

Solar: Towards a Shared-Everything Database on Distributed Log-Structured Storage

Tao Zhu[1], Zhuoyue Zhao[2], Feifei Li[2], Weining Qian[1], Aoying Zhou[1], Dong Xie[2]
Ryan Stutsman[2], Haining Li[3], Huiqi Hu[1,3]

[1]East China Normal University *[2]University of Utah* *[3]Bank of Communications*

Abstract

Efficient transaction processing over large databases is a key requirement for many mission-critical applications. Though modern databases have achieved good performance through horizontal partitioning, their performance deteriorates when cross-partition distributed transactions have to be executed. This paper presents Solar, a distributed relational database system that has been successfully tested at a large commercial bank. The key features of Solar include: 1) a shared-everything architecture based on a two-layer log-structured merge-tree; 2) a new concurrency control algorithm that works with the log-structured storage, which ensures efficient and non-blocking transaction processing even when the storage layer is compacting data among nodes in the background; 3) fine-grained data access to effectively minimize and balance network communication within the cluster. According to our empirical evaluations on TPC-C, Smallbank and a real-world workload, Solar outperforms the existing shared-nothing systems by up to 50x when there are close to or more than 5% distributed transactions.

1 Introduction

The success of NoSQL systems has shown the advantage of the scale-out architecture for achieving near-linear scalability. However, it is hard to support transaction in them, an essential requirement for large databases, due to the distributed data storage. For example, Bigtable [5] only supports single-row transactions, while others like Dynamo [6] do not support transactions at all. In response to the need for transaction support, NewSQL systems are designed for efficient OnLine Transaction Processing (OLTP) on a cluster with distributed data storage.

Distributed transaction processing is hard because of the need of efficient synchronization among nodes to ensure ACID properties and maintain good performance. Despite the significant progress and success achieved by many recently proposed systems [12, 27, 29, 19, 9, 23, 8, 37, 33], they still have various limitations. For example, the systems relying on shared-nothing architecture and 2PC (two-phase commit) heavily suffer from cross-partition distributed transactions, and thus require careful data partitioning with respect to given workloads. On the other hand, distributed shared-data systems like Tell [19] require specific hardware supports that are not commonly available yet at large scale.

That said, when no prior assumption can be made regarding the transaction workloads, and with no special hardware support, achieving high performance transaction processing on a commodity cluster is still a challenging problem. Meanwhile, prior studies have also shown it is possible to design high performance transaction engines on a single node by exploring the multi-core and multi-socket (e.g, NUMA) architecture. Both Silo [30] and Hekaton [7] have used a single server for transaction processing and demonstrated high throughput. However, such systems may not meet the needs of big data applications whose data cannot fit on a single node, hence requiring the support for a distributed data storage.

Inspired by these observations, our objective is to design a transactional database engine that *combines the benefits* of scalable data storage provided by a cluster of nodes and the simplicity for achieving efficient transaction processing on a single server node, *without* making any apriori assumptions on the transactional workloads *and without* requiring any special hardware support.

Bank of Communications, one of the largest banks in China, has faced these challenges. On one hand, new e-commerce applications from its own and its partners' mobile and online apps have driven the need for the support of ad-hoc transactions over large data, where little or no knowledge/assumptions can be made towards the underlying workloads as new apps emerge constantly. On the other hand, the bank has a strong interest towards better utilization of its *existing* hardware infrastructures to avoid costly new hardware investment if possible.

With that in mind, Solar is designed using a *shared-everything* architecture, where a server node (called *T-node*) is reserved for in-memory transaction processing and many storage nodes (called *S-nodes*) are used for data storage and read access. In essence, the S-nodes in Solar form a *distributed storage engine* and the T-node acts as *a main-memory transaction engine*. The distributed storage engine takes advantage of a cluster of nodes to achieve scalability in terms of the database capacity and the ability to service concurrent reads. The transaction engine provides efficient transaction processing and temporarily stores committed updates through its *in-memory committed list*. Periodically, *recently committed data items* on T-node are merged back into S-nodes through a *data compaction* procedure running in the background, *without* interrupting ongoing transactions. Overall, Solar is designed to achieve high performance transaction processing and scalable data storage.

To speed up update operations in the system, the in-memory committed list on T-node and the disk storage from all S-nodes *collectively form a distributed two-layer log-structured merge-tree* design [22]. Furthermore, a processing layer called *P-unit* is introduced to carry out *both* data access from S-nodes *and* any computation needed in a transaction so that the T-node can be freed from the burden of coordinating data access and performing business logic computation. This separation of storage and computation also enables the system to leverage all CPU resources for transaction scheduling and validation. Towards realizing the above design principle, we also design and implement a number of optimizations and algorithms to minimize the overhead in the system. Our contributions are summarized as follows:

- A distributed shared everything architecture with a T-node, a set of S-nodes and P-units is proposed for achieving high performance transaction processing.
- A hybrid concurrency control scheme called *MVOCC* is explored that combines the OCC (optimistic concurrency control) and the MVCC (multi-version concurrency control) schemes.
- A data compaction algorithm, as part of MVOCC, is designed to efficiently merge the *committed list* on T-node back to S-nodes periodically, *without interrupting* transaction processing on T-node.
- Several optimizations are investigated to improve the overall performance, e.g., separation of computation and storage through P-units, grouping multiple data access operations in one transaction, maintaining a bitmap for avoiding unnecessary data access to the distributed storage engine.

In our empirical evaluation on TPC-C, Smallbank and a real-world workload, Solar outperforms existing shared-nothing systems by 50x when the transactions requiring distributed commits are close to or more than 5%.

2 Solar Architecture

Solar is a distributed shared-everything relational database that runs concurrent transactions on a cluster of commodity servers. Figure 1 shows its architecture.

2.1 Design considerations

Shared-everything architecture. Shared-nothing architecture [12, 27] places data in non-overlapping partitions on different nodes in the hope that it can avoid expensive 2PC among nodes when almost all of the transactions only need to touch data on one partition and thus can run independently. For distributed transactions, multiple partitions with data involved need to be locked, blocking all other transactions that need to touch those partitions, which greatly increases system latency. Even worse, it only takes merely a handful of distributed transactions to always have locks on all the partitions and, as a result, system throughput can be reduced to nearly zero.

Instead, Solar employs a shared-everything architecture, where a transaction processing unit can access any data. ACID can be enforced at a finer granularity of individual records rather than at partitions. It also avoids expensive 2PC by storing updates on a single high-end server, enabling a higher transaction throughput.

In-memory transaction processing and scalable storage. Traditional disk-based databases rely on buffering mechanisms to reduce the latency of frequent random access to the data. However, this is several magnitudes slower than accessing in-memory data due to the limited size of the buffers and complication added to recovery.

In-memory transaction processing proves to be much more efficient than disk-based designs [7, 12]. Limited memory is always a key issue with in-memory transaction processing. Databases must have mechanisms to offload data to stable storage to free up memory for an unbounded stream of transactions. A key observation is that transactions typically only touch a very small subset of the whole database, writing a few records at a time in a database of terabytes of data. Thus, Solar reduces transaction processing latency by writing completely in memory while having an unbounded capacity by storing a consistent snapshot on a distributed disk-based storage, which can be scaled out to more nodes if needed.

Fine-grained data access control. In Solar, processing nodes directly access data stored in remote nodes via network, which can lead to overheads. Existing studies have shown that it is advantageous to use networks such as InfiniBand and Myrinet [19]. However, they are far from widely available. They require special software and hardware configurations. It is still unclear how to do that on a cluster of hundreds of off-the-shelf machines.

Solar is designed to work on a cluster of commodity servers, and thus uses a standard networking infrastructure based on Ethernet/IP/TCP. But network latency is significant because of the transition and data copying into and out of kernel. It also consumes more CPU than Infiniband, where data transport is offloaded onto NIC. To address the issue, we designed fine-grained data access to reduce network overhead, including caching, avoiding unnecessary reads and optimizing inter-node communication via transaction compilation. Fine-grained data access brings the transaction latency on par with the state-of-the-art systems and improves throughput by 3x.

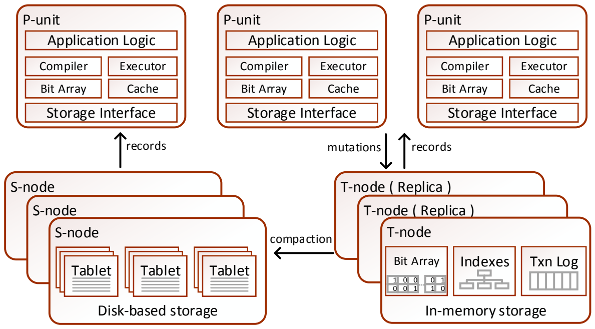


Figure 1: Architecture of Solar

2.2 Architecture overview

Figure 1 provides an overview of Solar's architecture. Solar separates transaction processing into *computation, validation and commit* phases using a *multi-version optimistic concurrency control protocol*. A transaction can be initiated on any one of the P-units, which do not store any data except several small data structures for data access optimization (Section 4). The P-unit handles all the data fetches from either T-node or S-nodes as well as transaction processing. The writes are buffered at the P-unit until the transaction commits or aborts. When the transaction is ready to commit, the P-unit sends the write set to T-node for validation and commit. Once T-node completes the validation, it writes the updates to its in-memory storage, and also a Write-Ahead Log to ensure durability. Finally, T-node notifies the P-unit if the transaction is successfully committed. P-units can be instantiated on any machine in or outside the cluster (typically on S-nodes or at client side). They offload most of the computation burden from T-node so that T-node can be dedicated to transaction management. Cluster information (e.g. states of all nodes, data distribution) are maintained by a manager node, and cached by other nodes.

Solar adopts a two-layer distributed storage that mimics the *log-structured merge tree (LSM-tree)* [22]. The storage layer consists of 1) a consistent database snapshot; and 2) all committed updates since the last snapshot. The size of the snapshot can be arbitrarily large and thus is stored in a distributed structure called SSTable across the disks of the S-nodes. Records in a table are dynamically partitioned into disjoint ranges according to their primary keys. Each range of records is stored in a structure called tablet (256 MB in size by default), which is essentially a B-tree index. The committed updates are stored in Memtable on T-node, which are from recent transactions and are typically small enough to fit entirely in memory. Memtable contains both a hash index and a B-tree index on the primary keys. The data entry points to all the updates (updated columns only) since the last snapshot, sorted by their commit timestamp. To access a specific record, a P-unit first queries Memtable. If there's no visible version in Memtable, it then queries SSTable for the version from the last snapshot.

The size of Memtable increases as transactions are committed. When it reaches certain memory threshold or some scheduled off-peak time (e.g. 12:00 am - 4:00 am for Bank of Communications), Solar performs a data compaction operation to merge the updates in Memtable into SSTable to free up the memory on T-node. At the end of data compaction, a new consistent snapshot is created in SSTable and Memtable drops all the committed updates prior to the start of the data compaction.

During data compaction, a new Memtable is created to handle new transactions arriving after the start of the data compaction. Then the old Memtable is merged into SSTable in a way similar to LSM-tree, namely merging two sorted lists from the leaf level of B-Tree index. Instead of overwriting the data blocks with new contents, we make new copies and apply updates on the copies. As we will explain in Section 3, transactions that have already started at the start of data compaction might still need to access the old SSTable. Thus, this approach minimizes the interruption to ongoing transactions.

Note that the function of T-node is twofold: it works as a transaction manager that performs timestamp assignment, transaction validation as well as committing updates; on the other hand, it serves as the *in-memory portion of the log-structured storage layer*. This architecture allows low-latency and high-throughput insertion, deletion and update through the in-memory portion. The log-structured storage also enables fast data compaction, which has a very small impact on the system performance because it mainly consumes network bandwidth instead of T-node's CPU resource.

Finally, Solar uses data replication to provide high availability and resistance to node failures. In SSTable, each tablet has at least 3 replicas and they are assigned to different S-nodes. Replication also contributes to achieve better load balancing among multiple S-nodes: a read request can access any one of the replicas. Memtable is replicated on two backup T-nodes. Details of data replication and node failures are discussed in Section 3.2.

3 Transaction Management

Solar utilizes both Optimistic Concurrency Control (OCC) and Multi-Version Concurrency Control (MVCC) to provide *snapshot isolation* [2]. Snapshot isolation is widely adopted in real-world applications, and many database systems (e.g. PostgreSQL prior to 9.1, Tell [19]) primarily support snapshot isolation, although it admits the write-skew anomaly that is prevented by serializable isolation. This paper focuses on Solar's support for snapshot isolation, and leave the discussion of serializable isolation to a future work. To ensure durability and support system recovery, redo log entries are persisted into the durable storage on T-node before transaction commits (i.e., write-ahead logging).

3.1 Supporting snapshot isolation

Solar implements snapshot isolation through combining OCC with MVCC [15, 2]. More specifically, MVOCC is used by T-node over Memtable. Recall that each record in Memtable maintains multiple versions. A transaction t_x is allowed to access versions created before its start time, which is called the *read-timestamp* and can be any timestamp before its first read. At the commit time, a transaction obtains a *commit-timestamp*, which should be larger than any existing *read-timestamp* or *commit-timestamp* of other transactions. Transaction t_x should also verify that no other transactions ever write any data, between t_x's *read-timestamp* and *commit-timestamp*, that t_x has also written. Otherwise, t_x should be aborted to avoid lost-update anomaly [2]. When a transaction is allowed to commit, it updates a record by creating a new version tagged with its *commit-timestamp*.

With MVOCC, SSTable contains, for all records in the database, the latest versions created by transactions with *commit-timestamps* are smaller than the *last data compaction time* (*compaction-timestamp*). Memtable contains newer versions created by transactions with *commit-timestamps* larger than *compaction-timestamp*.

T-node uses a global, monotonically increasing, counter to allocate timestamps for transactions. Transaction processing in Solar is decomposed into three phases: processing, validating and writing/committing.

Processing. In the processing phase, a worker thread of a P-unit executes the user-defined logic in a transaction t_x and reads records involved in t_x from both T-node and S-nodes. A transaction t_x obtains its *read-timestamp* (rt_x for short) when it first communicates with T-node. The P-unit for processing t_x reads the latest version of each record involved in t_x, whose timestamp is smaller than rt_x. In particular, it first retrieves the latest version from Memtable. If a proper version (i.e., timestamp less than rt_x) is not fetched, it continues to access the corresponding tablet of SSTable to read the record. During this process, t_x buffers its writes in a local memory space on the P-unit. When t_x has completed all of its business logic code, it enters the second phase. The P-unit sends a commit request for t_x containing t_x's write-set to T-node. T-node would then validate and commit the transaction.

Validating. The validation phase is conducted on T-node, which aims to identify potential write-write conflicts between t_x and other transactions. During the validation phase, T-node attempts to lock all records in t_x's write-set (denoted as w_x) on Memtable and checks, for any record $r \in w_x$, whether there is any newer version of r in Memtable whose timestamp is larger than rt_x. When all locks are successfully held by t_x and no newer version for any record in w_x is found, T-node guarantees that t_x has no write-write conflict and can continue to commit. Otherwise, T-node will abort t_x due to the lost update anomaly. Hence, after validation, T-node determines whether to commit or abort a transaction t_x. If it decides to abort t_x, T-node sends the abort decision back to the P-unit who sent in the commit request for t_x. The P-unit will simply remove the write-set w_x. Otherwise, the transaction t_x continues to the third phase.

Writing/Committing. In this phase, a transaction t_x first creates a new version for each record from its write-set w_x in Memtable, and temporarily writes its transaction ID x into the header field of each such record. Next, T-node obtains a *commit-timestamp* for t_x by incrementing the global counter. T-node then replaces the transaction identifier with t_x's *commit-timestamp* for each record with transaction ID x in Memtable (i.e., those from w_x). Lastly, T-node releases all locks held by t_x.

Correctness. Given a transaction t_x with *read-timestamp* (rt_x) and *commit-timestamp* (ct_x), Solar guarantees that t_x reads a consistent snapshot of the database and there is no lost update anomaly.

Consistent snapshot read: Firstly, t_x sees the versions written by all transactions committed before rt_x because those transactions have finished creating new versions for their write-sets and obtained their commit-timestamps before t_x is assigned rt_x as its read-timestamp. Secondly, the remaining transactions in the system always write a new data version using a commit-timestamp that is larger than rt_x. Hence, their updates will not be read by t_x. Hence, t_x always operates on a consistent snapshot.

Prevention of Lost Update: Lost update anomaly happens when a new version of record r is created by another transaction for $r \in w_x$, and the version's timestamp is in the range of (rt_x, ct_x). Assume the version is created by t_y. There are two cases:

1) t_y acquired the lock on record r prior to t_x's attempt to lock r. Thus, t_x only gets the lock after t_y has committed and created a new version of r. Hence, t_x will see the newer version of r during validation and be aborted.

2) t_y acquires the lock on r after t_x has secured the lock. In this case, t_y will not be able to obtain a commit timestamp until it has acquired the lock released by t_x, which means $ct_y > ct_x$. This contradicts with the assumption that the new version of r has a timestamp within (rt_x, ct_x). Recall that the timestamp of a new version for a record $r \in w_y$ is assigned the commit-timestamp of t_y.

3.2 System recovery

Failure of P-unit. When a P-unit fails, a transaction may still be in the processing phase if it has not issued the commit request. Such a transaction is treated as being aborted. For transactions in either the validation or the committing phase, they can be terminated by T-node without communicating with the failed P-unit. T-node will properly validate a transaction in this category and decide whether to commit or to abort. Both the snapshot isolation and durability are guaranteed, and all affected transactions are properly ended after a P-unit fails.

Failure of T-node. T-node keeps its Memtable in main memory. To avoid data loss, it uses WAL and forces redo log records to its disk storage for all committed transactions. When T-node fails, it is able to recover committed data by replaying the redo log. Moreover, to avoid being the single point of failure, Solar also synchronizes all redo log records to two replicas of T-node using a primary-backup scheme. Each replica catches up the content of T-node by replaying the log. When the primary T-node crashes, all actively running transactions are terminated; and further transaction commit requests are redirected to a secondary T-node quickly. As a result, Solar is able to recover from T-node failure and resume services in just a few seconds.

Failure of S-node. An S-node failure does not lead to loss of data as an S-node keeps all tablets on disk. The failure of a single S-node does not negatively impact the availability of system because all tablets have at least three replicas on different S-nodes. When one S-node has crashed, a P-unit can still access records of a tablet from the copy on another S-node.

3.3 Snapshot isolation in data compaction

Data compaction recycles memory used for Memtable. It produces a new SSTable by merging the current Memtable from T-node into the SSTable on S-nodes.

Data compaction. Let m_0 and s_0 be the current Memtable and SSTable respectively. Data compaction creates a new SSTable s_1 by merging m_0 and s_0. An empty Memtable m_1 replaces m_0 on T-node to service future transaction writes. Note that s_1 contains the latest version of each record originally stored in either m_0 or s_0, and is a consistent snapshot of the database. It indicates that there is a timestamp t_{dc} for the start of compaction such that transactions committed before t_{dc} store their updates in s_1 and transactions committed after t_{dc} keep new versions in m_1.

When data compaction starts, T-node creates m_1 for servicing new write requests. A transaction is allowed to write data into m_0 *if and only if* its validation phase occurred before data compaction started. T-node waits till all such transactions have committed (i.e., no more transaction will update m_0 any more). At this point, S-nodes start to merge m_0 with their local tablets. An S-node does not overwrite an existing tablet directly. Rather, it writes the new tablet using the copy-on-write strategy. Thus, ongoing transactions can still read s_0 as usual. An S-node acknowledges T-node when a tablet on that S-node involving some records in m_0 is completely merged with the new versions of those records from m_0. Data compaction completes when all new tablets have been created. T-node is now allowed to discard m_0 and truncate the associated log records.

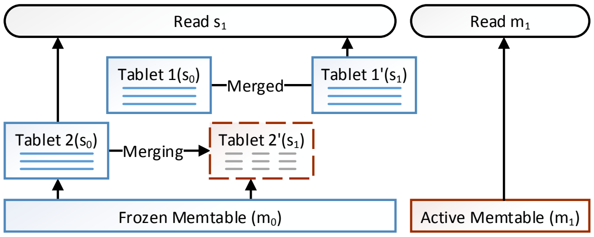


Figure 2: Data access during data compaction.

Figure 2 illustrates how to serve read access during data compaction. A read request for any newly committed record versions (after t_{dc}) is served by m_1; otherwise it is served by s_1. There are two cases when accessing s_1: if the requested record is in a tablet that has completed the merging process, only the new tablet in s_1 needs to be accessed (e.g., Tablet 1' in Figure 2); if the requested record is in a tablet that is still in the merging process (e.g., Tablet 2 in Figure 2), we need to access both that tablet from s_o and m_0.

Concurrency control. Snapshot isolation needs to be upheld during data compaction. The following concurrency control scheme is enforced. 1) If a transaction starts its validation phase before a data compaction operation is initiated, it validates and writes on m_0 as described in Section 3.1. 2) A data compaction operation can acquire a timestamp t_{dc} only when each transaction that started validation before the data compaction operation is initiated either aborts or acquires a commit-timestamp. 3) The data compaction can actually be started once all transactions with a commit-timestamp smaller than t_{dc} finish. 4) If a transaction t_x starts its validation phase after a data compaction operation is initiated, it can start validation only after the data compaction operation obtains its timestamp t_{dc}. The transaction t_x validates against *both* m_0 and m_1 but *only writes* to m_1. During validation, t_x acquires locks on both m_0 and m_1 for each record in its write set w_x, and verifies that no newer version is created relative to t_x's read-timestamp. Once passing validation, t_x writes its updates into m_1, after which t_x is allowed to acquire its commit-timestamp. 5) If a transaction acquires a read-timestamp which is larger than t_{dc}, it validates against and writes to m_1 only.

Correctness. Consistent snapshot read is guaranteed by assigning a read-timestamp to each transaction. Its correctness follows the same analysis as discussed for the normal transaction processing. The above procedure also prevents lost update during data compaction. Consider a transaction t_x with read timestamp rt_x and commit-timestamp ct_x. Assume that another transaction t_y exists, which has committed between rt_x and ct_x, i.e., $rt_x < ct_y < ct_x$, and t_y has written some records that t_x will also write later after t_y has committed. We only need to consider the case where $ct_y < t_{dc} < ct_x$, since, otherwise, lost update anomaly is guaranteed not to happen because both t_x and t_y will validate against the same set of Memtables (m_0 and/or m_1). This leads to the situation where $rt_x < ct_y < t_{dc} < ct_x$. Thus, t_x will be validated against both m_0 and m_1, and it will guarantee to see the committed updates made by t_y. As a result, t_x will be aborted since it will find at least one record with timestamp greater than its read timestamp rt_x. Hence, lost update anomaly still never happens even when data compaction runs concurrently with other transactions.

Recovery. The recovery mechanism is required to correctly restore both m_0 and m_1 when a node fails during an active data compaction. Data compaction acts as a boundary for recovery. Transactions committed before the start of the latest data compaction (that was actively running when a crash happened) should be replayed into m_0 while those committed after that should be replayed into m_1. Furthermore, we do *not* need to replay any transactions committed before the completion of the latest *completed* data compaction, since they have already been successfully persisted to SSTable through the merging operation of that completed data compaction. To achieve that, a compaction start log entry (CSLE) is persisted into the log on disk storage, when a data compaction starts, to document its t_{dc}. A compaction end log entry (CELE) is persisted when a data compaction ends with its t_{dc} serving as a unique identifier to identify this data compaction.

That said, failure of any P-unit does not lead to data loss or impact data compaction. When T-node fails, the recovery procedure replays the log from the CSLE with timestamp t_{dc}, which can be found in the last CELE. Initially, it replays the log into the Memtable m_0. When a CSLE is encountered, it creates a new Memtable m_1 and replays subsequent log entries into m_1. The merging into S-nodes continues after m_0 is restored from the recovery.

If an S-node fails during a data compaction, no data is lost since S-nodes use disk storage. But an S-node β may still be in the process of creating new tablets when it fails. Thus, when β recovers and rejoins the cluster, it contains the tablets of old SSTable and incomplete tablets produced during merging. If the system has already completed the data compaction (using other replicas for the failed node), there is at least one replica for each tablet in the new SSTable. The recovered node β simply copies the necessary tablets from a remote S-node. If data compaction has not completed, β would continue merging by reading records in m_0 from T-node.

Storage management. During data compaction, m_0 and s_0 (the existing SSTable before the current compaction starts) remain read only while s_1 and m_1 are being updated. When compaction completes, m_0 and s_0 are to be truncated. But they can only be truncated when no longer needed for any read access. In summary, m_0 and s_0 can be truncated when the data compaction has completed *and* no transaction has a read timestamp smaller than t_{dc}.

4 Optimization

It is important for Solar to reduce the network communication overhead among P-units, S-nodes and T-node. To achieve better performance, we design fine-grained data access methods between P-units and the storage nodes.

4.1 Optimizing data access

The correct data version that a transaction needs to read is defined by the transaction's read-timestamp, which could be stored either in SSTable on S-nodes or in Memtable on T-node. Thus, Solar does not know where a record (or columns of a record) should be read from, and P-units have to access both SSTable on S-nodes and Memtable on T-node to ensure read consistency (though one of which will turn out to be an incorrect version).

Here, we first present an SSTable cache on P-units to reduce data access between P-units and S-nodes. Then, an asynchronous bit array is designed to help P-units identify potentially useless data accesses to T-node.

4.1.1 SSTable cache

A P-unit needs to pull records from SSTable. These remote data accesses can be served efficiently using a data cache. The immutability of SSTable makes it easy to build a cache pool on a P-unit. The cache pool holds records fetched from SSTable and serves data accesses to the same records.

The cache pool is a simple *key-value store*. The *key* stores the primary key and the *value* holds the corresponding record. All entries are indexed by a hash map. A read request on a P-unit first looks for the record from its cache pool. Only if there is a cache miss, the P-unit pulls the record from an S-node and adds it to its cache pool. The cache pool uses a standard buffer replacement algorithm to satisfy a given memory budget constraint.

Since SSTable is immutable and persisted on disk, Solar does not persist the cache pools. Entries in a cache pool do expire when the SSTable they were fetched from is obsolete after a data compaction operation. A P-unit builds a new cache pool when that happens.

4.1.2 Asynchronous bit array

SSTable is a consistent snapshot of the whole database. In comparison, Memtable only stores the newly created data versions after the last data compaction, which must be a small portion of the database. As a result, most likely a read request sent to a T-node would fetch nothing from T-node. We call this phenomenon *empty read*. These requests are useless and have negative effects. They increase latency and consume T-node's resources.

To avoid making many empty reads, T-node uses a succinct structure called *memo structure* to encode the existence of items in Memtable. The structure is periodically synchronized to all P-units. Each P-unit examines its local memo to identify potential empty reads.

The memo structure is a bit array. In the bit array, each bit is used to represent whether a column of a tablet has been modified or not. That is to say, if any record of a tablet T has its column C modified, the bit corresponding to (T,C) is turned on. Otherwise, the bit is turned off. Other design choices are possible, e.g., to encode the record-level information, but that would increase the size of the bit-array dramatically.

Solar keeps two types of bit arrays. The first type is a real-time bit array on T-node, denoted as b. The second type is an asynchronous bit array on each P-unit, which is a copy of b at some timestamp t, denoted as $b' = b_t$ where b_t is the version of b at time t. A P-unit queries b' to find potential empty reads without contacting T-node.

On T-node, b is updated when a new version is created for any column of a record for the first time. Note that when a version is created for a data item (a column value) that already exists in Memtable, it is not necessary to update b, as that has already been encoded in b. Each P-unit pulls b from T-node periodically to refresh and synchronize its local copy b'.

During query processing for a transaction t_x on a P-unit p, p examines its local b' to determine whether T-node contains newer versions for the columns of interest of any record in t_x's read set. If (T,C) is 0 in b' for such a column C, p treats the request as an empty read and does not contact T-node; otherwise, p will send a request to pull data from T-node.

Clearly, querying b' leads to false positives due to the granularity of the encoding, and such false positives will lead to empty reads to T-node. Consider in tablet T, row r_1 has its column C updated and row r_2 has not updated its column C. When reading column C of r_2, a P-unit may find the bit (T,C) in b' is set while there is no version for $r_2.C$ on T-node. In fact, the above method is most effective for read-intensive or read-only columns. They seldom have their bits turned on in the bit array.

Querying b' may also return false negatives because it is not synchronized with the latest version of b on T-node. Once a false negative is present, a P-unit may miss the latest version of some values it needs to read and end up using an inconsistent snapshot. To address this issue, a transaction must check all potential empty reads during its validation phase. If a transaction sees the bit for (T,C) is 0 in b' during processing, it needs to check whether the bit is also 0 in b during validation. If any empty read previously identified by b' cannot be confirmed by b, a transaction has to be re-processed by reading the latest versions in Memtable. False negatives are rare because b does not see frequent update: it is only updated at the first time any row in tablet T has its column C modified.

4.2 Transaction compilation

Solar supports JDBC/ODBC connections, as well as stored procedures. The latter takes the one-shot execution model [26] and avoids client-server interaction. This poses more processing burden on the DBMS, but enables server-side optimizations [33, 39, 40]. Solar designs a compilation technique to optimize inter-nodes communication by generating an optimized physical plan.

read	Memtable read
	SSTable read
write	update local buffer on P-unit **(local operation)**
process	expression, project, sort, join ... **(local operation)**
compound	loop, branch

Table 1: Possible operations in a physical plan.

Execution graph. The physical plan, to be executed by a P-unit, of a stored procedure is represented as a sequence of operations in Table 1 (nested structures, such as branch or loop, can be viewed as a compound operation). Reads are implemented via RPC while write and process/computation are *local operations*. Hence, reads are the key to optimizing network communication.

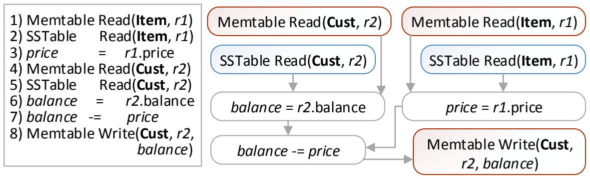

1) Memtable Read(**Item**, *r1*)
2) SSTable Read(**Item**, *r1*)
3) *price* = *r1*.price
4) Memtable Read(**Cust**, *r2*)
5) SSTable Read(**Cust**, *r2*)
6) *balance* = *r2*.balance
7) *balance* -= *price*
8) Memtable Write(**Cust**, *r2*, *balance*)
Memtable Read(**Cust**, *r2*)
Memtable Read(**Item**, *r1*)
SSTable Read(**Cust**, *r2*)
SSTable Read(**Item**, *r1*)
balance = *r2*.balance
price = *r1*.price
balance -= *price*
Memtable Write(**Cust**, *r2*, *balance*)

Figure 3: Example of operation sequence and execution graph.

Two operations have to be executed in order if they have 1) *procedure constraint*: two operations contain data/control dependence [21]; or 2) *access constraint*: two operations access the same record and one operation is a write. In practice, we cannot always determine two database records are the same during compilation, so we treat it as a potential access constraint if two operations are accessing the same table. Then, we can represent an operation sequence as an *execution graph*, where the nodes are operations and edges are the constraints and represent the execution order (Figure 3).

We also support branches and loops as compound operations. A compound operation is a complex operation if it contains multiple reads. If it only contains one read, the compound operation is viewed as the same type of read (defined in Table 1) as that single read. Otherwise, a compound operation is viewed as a local operation. We adopt loop distribution [14] to split a large loop into multiple smaller loops so that they can be categorized more specifically. For a read operation in a branch block, it can be moved out for speculative execution since reads do not have side effect and thus are safe to execute *even if* the corresponding branch is not taken.

Grouping Memtable **Reads.** To reduce the number of RPCs to T-node, we can group multiple Memtable reads together in one RPC to T-node if they do not have constraints between them. This is done in two passes over the physical plan. The first pass finds all the Memtable reads not constrained by any other reads via a BFS over the execution graph. The second pass starts from the unconstrained Memtable reads and marks all *local operations* that precede them. Before executing transaction logics, all those local operations marked in pass 2 get executed first. Then the Memtable reads marked in pass 1 are sent in a single RPC request to T-node.

Pre-executing SSTable **Reads.** SSTable reads can be issued even before a transaction obtains its read-timestamp from T-node and we concurrently execute them with other operations, since there is only one valid snapshot in SSTable at a time. This requires that the SSTable reads are not constrained by other operations. During execution, the result of a SSTable read might or might not be used depending on if there is update to the same record in Memtable. Though this optimization might introduce unused SSTable reads, the problem can be mitigated by the SSTable cache pool. The main benefit of pre-executing SSTable reads is reducing wait time and thus reducing latency. The SSTable reads that can be pre-executed can be found using a similar algorithm to the one that finds Memtable reads that can be grouped.

Remarks. The optimizations described in this section are desgined for short transactions. Other workloads, such as bulk loading, OLAP, require additional optimizations. For bulk loading, it is possible to skip the T-node and directly load data into S-nodes. For OLAP queries, they can be executed upon a consistent database snapshot, and some relational operators can be pushed down into storage nodes to reduce inter-node data exchange.

5 Experiment

We implemented Solar by extending the open-sourced version of Oceanbase (OB) [1]. In total, 58,281 lines were added or modified on its code base. Hence, Solar is a full-fledged database system, implemented in 457,206 lines of C++ code. In order to compare it to other systems that require advanced networking infrastructures, we conducted all experiments using 11 servers on Emulab [38], which allows configuring different network topologies and infrastructures. Each server has two 2.4 GHz 8-Core E5-2630 processors (32 threads in total when hyper-threading enabled) and 64GB DRAM, *connected through a 1 Gigabits Ethernet by default. By default ten servers* are used to deploy the database system. One server is used to simulate clients. We compared Solar with MySQL-Cluster 5.6, Tell (shared-everything) [19], and VoltDB Enterprise 6.6 (shared-nothing) [27]. Though Tell is designed for InfiniBand, we used a simulated InfiniBand over Ethernet to have a fair comparison. We use Tell-1G (Tell-10G) to represent the Tell system using 1-Gigabits (10-Gigabits) network respectively.

Solar is not compared with lightweight prototype systems that aim at verifying the performance of new concurrency control scheme, such as Silo [30]. These systems achieve impressive throughput, but their implementations lack many important features, such as durable logging, disaster recovery and a SQL engine. These features often introduce significant performance overhead, but are ignored by these lightweight system prototypes.

Solar deploys the T-node on a single server. It deploys both an S-node and a P-unit on each of the remaining nodes. Tell deploys a commit manager on a single server. It uses two servers for storage node deployment and the rest for processing nodes. We tested different combinations of processing node and storage node instances and chose the best configuration. Tell uses more processing node instances and fewer storage nodes. MySQL-Cluster deploys both a mysqld and a ndbmtd instance on each server. VoltDB creates 27 partitions on each server, which is based on the officially recommended strategy [32]. It was determined by adjusting partition numbers to achieve the best performance on a single server.

We used three different benchmarks. Performance of different systems are evaluated by transaction processed per second (TPS). In each test instance, we adjusted the number of clients to get the best throughput.

5.1 TPC-C benchmark

We use a standard TPC-C workload with 45% NewOrder, 43% Payment, 4% OrderStatus, 4% Delivery and 4 % StockLevel requests. Request parameters are generated according to the TPC-C specification. *By default, 200 warehouses* are populated in the database. Warehouse keys are used for horizontal partitioning. Initially, Solar stores 1.6 million records (2.5 GB) in the Memtable and 100 million records (42GB) in the SSTable (with 3x replication enabled). After the benchmark finishes, there are 11 GB data in the Memtable and the size of SSTable is about 655 GB.

Figure 4 shows the performance of different systems when we vary the number of warehouses. Solar achieves about 53k TPS on 50 warehouse, and increases to about 75k TPS with 350 warehouses. When more warehouses are populated, there are less access contention in the workload, leading to fewer conflicts and higher concurrency. Solar clearly outperforms the other systems. Its throughput is 4.8x of that of Tell-10G (about 15.6k TPS) with 350 warehouses. Note that Tell-1G, which uses the same network infrastructure as Solar, performs even worse than Tell-10G. VoltDB exhibits the worst performance due to distributed transactions. Lastly, Oceanbase is primarily designed for processing very short transactions and thus is inefficient on general transaction workloads. Solar always achieves at least 10x throughput improvement over Oceanbase. Therefore, we skip Oceanbase in other benchmarks.

Figure 5 evaluates the scalability when using different number of nodes. The throughputs of Solar, Tell and MySQL-Cluster increase with more nodes. In contrast, the throughput of VoltDB deteriorates for the following

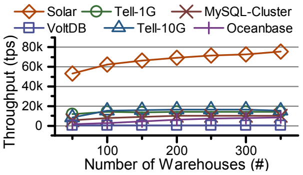


Figure 4: TPC-C: vary number of warehouses.

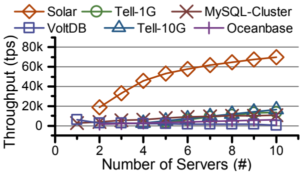


Figure 5: TPC-C: vary number of servers.

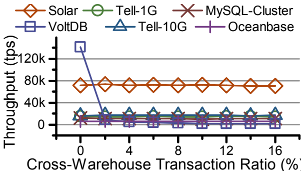


Figure 6: TPC-C: vary ratio of cross-warehouse transactions.

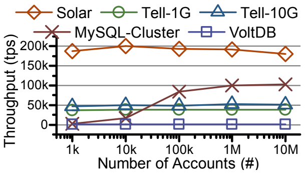


Figure 7: Smallbank: vary number of accounts.

reason. Distributed transactions are processed by a single thread in VoltDB. They block all working threads of the system. With more servers being used, it becomes more expensive for such request to be processed. The throughput growth in Solar slows down with more than 7 servers. As there are more access conflicts with a higher number of client requests, more transactions fail in the validation phase. Another reason is that T-node receives more loads when working with more P-units, *and* in our experimental setting, T-node uses the same type of machine as that used for P-units. Hence, the overall performance increases sub-linear with the number of P-units. However, in the real deployment of Solar, a high-end server is recommended for T-node, whereas P-units (and S-nodes) can be served with much less powerful machines.

Figure 6 shows the results when we vary the ratio of cross-warehouse transactions. If a transaction accesses records from multiple warehouse, it is a distributed transaction. VoltDB achieves the best performance (141k TPS) when there are no distributed transactions, which is about 2.0x of Solar (about 70k TPS). But as the ratio of distributed transactions increase, VoltDB's performance drops drastically as it uses horizontal partitioning to scale out. The other systems are not sensitive to this ratio.

Latency(ms)	Solar	Tell -1G	Tell -10G	MySQL-Cluster	VoltDB	OB
Payment	6	17	7	17	15619	38
NewOrder	15	28	12	103	30	60
OrderStatus	6	20	8	23	14	30
Delivery	40	160	53	427	14	174
StockLevel	9	14	7	17	14	60
Overall	12	30	12	95	2751	54

Table 2: 90th Latency, TPC-C workload.

Table 2 lists the 90th latency. Solar has a short latency for each transaction. Tell benefits from the better network. It gets better latency with the 10-Gbit network than the 1-Gbit one. The long latency of MySQL-Cluster comes from the network interaction between the database servers and clients because it uses JDBC instead of stored procedures. VoltDB is slow on distributed transactions. Under the standard TPC-C mix, about 15.0% Payment and 9.5% NewOrder requests are distributed transactions. Hence, the 90th latency of Payment is long. Though the 90th latency of NewOrder is small, its 95th latency reaches 15,819 ms.

5.2 Smallbank benchmark

Smallbank simulates a banking application. It contains 3 tables and 6 types of transactions. The workload contains 15% Amalgamate transactions, 15% Balance transactions, 15% DepositChecking transactions, 25% SendPayment transactions, 15% TransactSavings transactions and 15% WriteCheck transactions. Amalgamate and SendPayment operate on two accounts at a time. The other transactions access only a single account. We populated 10 million users into the database. Initially, there are 8M records (3 GB) in Memtable and 30M records (1.1 TB) in SSTable. After execution, Memtable has 5.2 GB data, and SSTable has about 1.1 TB data.

Figure 7 evaluates different systems by populating different number of accounts in the database. Note that *x-axis is shown in log-scale*. Solar has the best overall performance. Its throughput initially increases as the number of accounts increases, because less contention when there are more accounts. Due to the drop of SSTable's cache hit ratio as the number accounts further increases to 10M, P-units need to issue remomte data access to S-nodes. As a result, its throughput slightly drops.

Latency(ms)	Solar	Tell-1G	Tell-10G	MySQL-Cluster	VoltDB
Amalgamate	5	5	4	8	100
Balance	3	3	3	4	5
Deposit	4	4	4	3	5
SendPayment	7	4	4	12	102
Xact Savings	3	4	4	6	5
WriteCheck	5	4	4	5	6
Overall	5	4	4	8	92

Table 3: 90th Latency, Smallbank workload.

Tell shows a fairly stable performance, but 10G Ethernet only improves its throughput slightly. MySQL Cluster also has a better performance initially with more accounts, but stabilizes once it has maxed out all hardware resources. The performance of VoltDB is limited by cross-partition transactions. Table 3 lists the 90th latency number. It takes VoltDB much longer time than others to process *Amalgamate* and *SendPayment* and there are 40% such transactions in this workload.

Figure 8 evaluates each systems with different number of servers. Here, we populated 1M accounts in the database. Solar shows the best performance and scalability with respect to the number of servers. The throughputs of Solar, Tell and MySQL-Cluster scale linearly with the number of servers. The throughput of VoltDB is still quite limited by distributed transaction processing.

5.3 E-commerce benchmark

E-commerce is a workload from an e-business client of Bank of Communications. It includes 7 tables and 5 transaction types. There are two user roles in this application: buyer and seller. There are 4 tables for buyers:

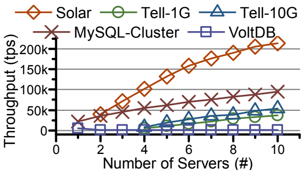


Figure 8: Smallbank: vary number of servers.

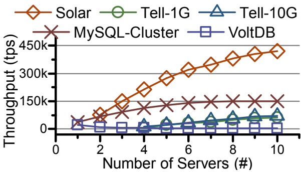


Figure 9: E-commerce: vary number of servers.

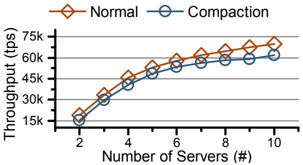


Figure 10: TPC-C: data compaction.

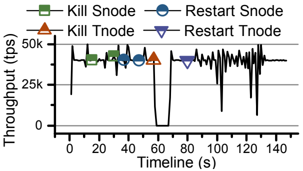


Figure 11: Solar: throughput under node failures.

User, *Cart*, *Favorite* and *Order* and 3 tables for sellers: *Seller*, *Item* and *Stock*. These tables are partitioned by *user_id* and *seller_id* respectively. At the start of the experiment, Solar has 11M records (5GB) in Memtable, and 25M records (815GB) in SSTable. When all experiments are completed, the Memtable has 8.6 GB data and the size of SSTable is 881GB.

The workload has 88% OnClick transactions, 1% AddCart transactions, 6% Purchase transactions and 5% AddFavorite transactions. The OnClick request is a *r*ead-only transaction while the others are *r*ead-write ones. OnClick reads an item and accesses *Item* and *Stock*. AddCart inserts an item into a buyer's cart and accesses *User* and *Cart*. AddFavorite inserts an item into a buyer's favorite list and updates the item's popular level. It accesses *User*, *Favorite* and *Item*. Purchase creates an order for a buyer and decrements the item's quantity. It accesses *User*, *Order*, *Item* and *Stock*.

Latency(ms)	Solar	Tell-1G	Tell-10G	MySQL-Cluster	VoltDB
OnClick	1	8	4	4	4
AddFavorite	2	12	5	6	47
AddCart	2	2	14	4	4
Purchase	4	12	4	6	49
Overall	1	8	4	4	19

Table 4: 90th Latency, E-commerce workload.

Figure 9 shows the performance of each system using different number of servers. The throughput of Solar increases with the number of servers used. It has achieved about 438k TPS when 10 servers are used, and is at least 3x that of any other system. As shown in Table 4 for the 90th latency, most transactions completed within 1ms by Solar. MySQL-Cluster and Tell also see performance improvement when more servers are used. But they have higher latency as shown in Table 4. VoltDB is highly inefficient on AddFavorite and Purchase because tables accessed by these transactions use different partition keys. These transactions may visit multiple partitions which block other single-partition transactions. As a result, OnClick and AddCart also have longer latency.

5.4 Data compaction

During transaction processing, Solar may initiate a data compaction in the background. Figure 10 shows the impact of data compaction on the performance, when Solar is processing the standard TPC-C workload. As shown in Figure 10, data compaction has little negative effect on the performance when 5 or less servers are used. It is because the performance is mainly limited by the number of P-units in these cases, and compaction would not influence the operation of P-units. When more servers are used, there is about 10% throughput loss. This is because at this point T-node has more impact on the overall system performance when more servers are introduced. Data compaction consumes part of the network bandwidth and CPU resources, which are also required by transaction processing on T-node.

5.5 Node failures

We next investigate the impact of node failures in Solar. In this experiment, 3 servers were used to deploy T-nodes, and 7 servers were used to deploy S-nodes and P-units. One T-node acts as the primary T-node, and the other two are secondary T-nodes. The TPC-C benchmark was used with 200 warehouses populated, and we terminated some servers at some point during execution. Figure 11 plots the changes of throughput against the time.

Removing 2 S-nodes does not impact the performance, as the SSTable keeps 3 replicas for each tablet and each P-unit also caches data from SSTable. Thus, losing 2 S-nodes does not influence performance. We then terminated the primary T-node. Immediately after it went down, the throughput drops to 0 because no T-node can service write requests now. After about 7 seconds, a secondary T-node becomes the primary and the system continues to function. After the failed T-node re-joins the cluster, the new primary T-node has to read redo log entries from the disk and send them to the T-node in recovery. Thus, the performance fluctuates and drops a little bit due to this overhead. It takes about 40 seconds for the failed T-node to catch up with the new primary, after which the system throughput returns to the normal level.

5.6 Access optimizations

Figure 12 evaluates the performance improvement brought by different access optimizations. The *y-axis shows the normalized performance to a baseline system without using any optimization.* The figure shows the improvement brought by enabling each individual optimization, as well as all of them, using the TPC-C workload. Other workloads share the similar performance trends. With more P-units and S-nodes deployed in system, the individual optimizations show different trends of improvement. The effectiveness of SSTable cache drops because the overall data access throughput increases when more S-nodes are deployed. However, the accesses to T-node are more contentious as more P-units communicate with the single T-node. With transaction compilation enabled, small data accesses to T-node are

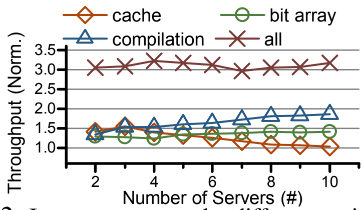


Figure 12: Improvements under different optimizations.

combined, which improves the overall throughput when there are more P-units. The bit array shows a relatively stable impact to the throughput because it prunes data access to T-node at the column level, which is related to the workload rather than the number of servers. As long as a column is not read-only in any row in a tablet, it cannot prune the data access to T-node. When all optimizations are used together, they bring about 3x throughput improvement regardless of the number of servers used.

6 Related Work

Single-node system. Single-node in-memory systems have exploited NUMA architectures, RTM, latch-free data structures, and other novel techniques to achieve high performance transaction processing, such as Silo [30], Hekaton [7], Hyper [13, 24], DBX [34], and others. The usage of these systems are subject to the main memory capacity on a single node as they require all data stored in the memory. Deuteronomy's [17] transaction component (TC) uses pessimistic, timestamp-based MVCC with decoupled atomic record stores. It can manage data sharded over multiple record stores, but Deuteronomy is not itself networked or distributed; instead stores are on different CPU sockets. It ships updates to the data storage via log replaying and all reads have to go through TC. In contrast, Solar uses MVOCC and a cluster of data storage, and it can potentially skip T-node access using its asynchronous bit arrays.

Shared-nothing systems. Horizontal partitioning is widely used to scale out. Examples include HStore [12, 26], VoltDB [27], Accordion [25], E-Store [28]. We have discussed their limitations in Section 2.1. Calvin [29] takes advantage of deterministic execution to maintain high throughput even with distributed transactions. However, it requires a separate reconnaissance query to predict unknown read/write set. Oceanbase [1] is Alibaba's distributed shared-nothing database designed for short transactions. In shared-nothing systems, locking happens at partition level. To get subpartition locking, distributed locks or a central lock manager must be implemented, which goes against the principle for strict partitioning (i.e., get rid of distributed locking/latching), and reintroduces (distributed) locking and latching coordination overheads and defeats the gains of shared nothing. That said, new concurrency control schemes can improve the performance of distributed transactions (e.g., [20]), when certain assumptions are made (e.g., knowing the workload apriori, using offline checking, deterministic ordering, and dependency tracking).

Shared-everything systems. The shared-everything architecture is an alternative choice to enable high scalability and high performance, where any node can access and modify any record in the system. Traditional shared-everything databases, like IBM DB2 Data Sharing [11] and Oracle RAC [4], suffer from expensive distributed lock management. Modern shared-everything designs exploit advanced hardware to improve performance, such as Tell [19], DrTM [37] and DrTM+B [36] (with live reconfiguration and data repartitioning), HANA SOE [10]. Solar, on the other hand, uses commodity servers and does not rely on special hardwares.

Log-structured storage. The log-structured merge tree [22] is optimized for insertion, update and deletion. It is widely adopted by many NoSQL vendors, such as LevelDB [18], BigTable [5], Cassandra [16] and etc. However, none of these supports multi-row transactions. LogBase [31] is a scalable log-structured database with a *log file only storage* where the objective is to remove the write bottleneck and to support fast system recovery, rather than optimizing OLTP workloads. Hyder II optimizes OCC for tree-structured, log-structured databases [3] which Solar may leverage for further improving its concurrency control scheme. vCorfu [35] implements materialized streams on a shared log to support fast random reads. But, it increases transaction latency as committing requires at least four network roundtrips.

7 Conclusion

This work presents Solar, a high performance and scalable relational database system that supports OLTP over a distributed log-structured storage. Extensive empirical evaluations have demonstrated the advantages of Solar compared to other systems on different workloads. Solar has been deployed at Bank of Communications to handle its e-commerce OLTP workloads. We plan to open source Solar on GitHub. Current and future works include designing a more effective query optimizer and task processing module, by leveraging the NUMA architecture, improving its concurrency control scheme, and designing an efficient and scalable OLAP layer.

Acknowledgments. Tao Zhu, Weining Qian and Aoying Zhou are supported by 863 Program (2015AA015307), National Key R&D Plan Project (2018YFB1003303), NSFC (61432006 and 61332006). Feifei Li, Zhuoyue Zhao and Dong Xie are supported in part by NSF grants 1619287 and 1443046. Feifei Li is also supported in part by NSFC grant 61729202. Ryan Stutsman is supported in part by NSF grant CNS-1750558. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The authors greatly appreciate the valueable feedback provided by USENIX ATC reviewers.

References

[1] ALIBABA OCEANBASE. Oceanbase. https://github.com/alibaba/oceanbase, 2015.

[2] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ANSI SQL isolation levels. In *SIGMOD* (1995), vol. 24, ACM, pp. 1–10.

[3] BERNSTEIN, P. A., DAS, S., DING, B., AND PILMAN, M. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *SIGMOD* (2015), pp. 1295–1309.

[4] CHANDRASEKARAN, S., AND BAMFORD, R. Shared cache-the future of parallel databases. In *ICDE* (2003), IEEE, pp. 840–850.

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *TOCS 26*, 2 (2008), 4.

[6] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007), vol. 41, ACM, pp. 205–220.

[7] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-A., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD* (2013), ACM, pp. 1243–1254.

[8] DRAGOJEVIC, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *NSDI* (2014), pp. 401–414.

[9] DRAGOJEVIC, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP* (2015), pp. 54–70.

[10] GOEL, A. K., POUND, J., AUCH, N., BUMBULIS, P., MACLEAN, S., FÄRBER, F., GROPENGIESSER, F., MATHIS, C., BODNER, T., AND LEHNER, W. Towards scalable real-time analytics: an architecture for scale-out of OLxP workloads. *PVLDB 8*, 12 (2015), 1716–1727.

[11] JOSTEN, J. W., MOHAN, C., NARANG, I., AND TENG, J. Z. DB2's use of the coupling facility for data sharing. *IBM Systems Journal 36*, 2 (1997), 327–351.

[12] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P., MADDEN, S., STONEBRAKER, M., ZHANG, Y., ET AL. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB 1*, 2 (2008), 1496–1499.

[13] KEMPER, A., AND NEUMANN, T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE* (2011), IEEE, pp. 195–206.

[14] KENNEDY, K., AND MCKINLEY, K. S. *Maximizing loop parallelism and improving data locality via loop fusion and distribution*. Springer, 1993.

[15] KUNG, H.-T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *TODS 6*, 2 (1981), 213–226.

[16] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *SIGOPS 44*, 2 (2010), 35–40.

[17] LEVANDOSKI, J., LOMET, D., SENGUPTA, S., STUTSMAN, R., AND WANG, R. High performance transactions in Deuteronomy. Conference on Innovative Data Systems Research (CIDR 2015).

[18] LEVELDB. LevelDB. http://leveldb.org/, 2017.

[19] LOESING, S., PILMAN, M., ETTER, T., AND KOSSMANN, D. On the design and scalability of distributed shared-data databases. In *SIGMOD* (2015), ACM, pp. 663–676.

[20] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting more concurrency from distributed transactions. In *OSDI* (2014), pp. 479–494.

[21] MUCHNICK, S. S. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[22] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[23] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., ET AL. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS 43*, 4 (2010), 92–105.

[24] RÖDIGER, W., MÜHLBAUER, T., KEMPER, A., AND NEUMANN, T. High-speed query processing over high-speed networks. *PVLDB 9*, 4 (2015), 228–239.

[25] SERAFINI, M., MANSOUR, E., ABOULNAGA, A., SALEM, K., RAFIQ, T., AND MINHAS, U. F. Accordion: elastic scalability for database systems supporting distributed transactions. *PVLDB 7*, 12 (2014), 1035–1046.

[26] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The end of an architectural era:(it's time for a complete rewrite). In *PVLDB* (2007), VLDB Endowment, pp. 1150–1160.

[27] STONEBRAKER, M., AND WEISBERG, A. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull. 36*, 2 (2013), 21–27.

[28] TAFT, R., MANSOUR, E., SERAFINI, M., DUGGAN, J., ELMORE, A. J., ABOULNAGA, A., PAVLO, A., AND STONEBRAKER, M. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *PVLDB* (2014), 245–256.

[29] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD* (2012), pp. 1–12.

[30] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *SOSP* (2013), pp. 18–32.

[31] VO, H. T., WANG, S., AGRAWAL, D., CHEN, G., AND OOI, B. C. Logbase: A scalable log-structured database system in the cloud. *PVLDB 5*, 10 (2012), 1004–1015.

[32] VOLTDB INC. VoltDB. https://www.voltdb.com/, 2017.

[33] WANG, Z., MU, S., CUI, Y., YI, H., CHEN, H., AND LI, J. Scaling multicore databases via constrained parallel execution. In *SIGMOD* (2016), ACM, pp. 1643–1658.

[34] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In *EuroSys* (2014), pp. 26:1–26:15.

[35] WEI, M., TAI, A., ROSSBACH, C. J., ABRAHAM, I., MUNSHED, M., DHAWAN, M., STABILE, J., WIEDER, U., FRITCHIE, S., SWANSON, S., FREEDMAN, M. J., AND MALKHI, D. vCorfu: A cloud-scale object store on a shared log. In *USENIX NSDI* (2017), pp. 35–49.

[36] WEI, X., SHEN, S., CHEN, R., AND CHEN, H. Replication-driven live reconfiguration for fast distributed transaction processing. In *USENIX ATC* (2017), pp. 335–347.

[37] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *SOSP* (2015), ACM, pp. 87–104.

[38] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *OSDI* (2002), pp. 255–270.

[39] WU, Y., CHAN, C.-Y., AND TAN, K.-L. Transaction healing: Scaling optimistic concurrency control on multicores. In *SIGMOD* (2016), ACM, pp. 1689–1704.

[40] YAN, C., AND CHEUNG, A. Leveraging lock contention to improve OLTP application performance. *PVLDB* (2016), 444–455.

Toward Coordination-free and Reconfigurable Mixed Concurrency Control

Dixin Tang
University of Chicago

Aaron J. Elmore
University of Chicago

Abstract

Recent studies show that mixing concurrency control protocols within a single database can significantly outperform a single protocol. However, prior projects to mix concurrency control either are limited to specific pairs of protocols (e.g mixing two-phase locking (2PL) and optimistic concurrency control (OCC)) or introduce extra concurrency control overhead to guarantee their general applicability, which can be a performance bottleneck. In addition, due to unknown and shifting access patterns within a workload, candidate protocols should be chosen dynamically in response to workload changes. This requires changing candidate protocols online without having to stop the whole system, which prior work does not fully address. To resolve these two issues, we present CormCC, a general mixed concurrency control framework with no coordination overhead across candidate protocols while supporting the ability to change a protocol online with minimal overhead. Based on this framework, we build a prototype main-memory multicore database to dynamically mix three popular protocols. Our experiments show CormCC has significantly higher throughput compared with single protocols and state-of-the-art mixed concurrency control approaches.

1 Introduction

With an increase in CPU core counts and main-memory capacity, concurrency control has become a new bottleneck in multicore main-memory databases due to the elimination of disk stalls [8, 34]. New concurrency control protocols and architectures focus on enabling high throughput by fully leveraging available computation capacity, while supporting ACID transactions. Some protocols try to minimize the overhead of concurrency control [10], while other protocols strive to avoid single contention points across many cores [27, 35]. However, these single protocols are typically designed for specific workloads, may only exhibit high performance under their optimized scenarios, and have poor performance in others. Consider H-Store's concurrency control protocol that uses coarse-grained exclusive partition locks and a simple single-threaded executor per partition [10]. This approach is ideal for partitionable workloads that have tuples partitioned such that a transaction is highly likely to access only one partition. But this approach suffers decreasing throughput with increasing cross-partition transactions [34, 27, 20]. Here, an optimistic protocol may be preferred if the workload mainly consists of read operations or a pessimistic protocol may be ideal if the workload exhibits high conflicts. An appealing solution to this tension is to combine different protocols such that each protocol can be used to process a part of workload that they are optimized for, and avoid being brittle to scenarios where single protocols suffer.

Efficiently mixing multiple concurrency control protocols is challenging in several ways. First, it should not be limited to a specific set of protocols (e.g. OCC and 2PL [22, 28]), but be able to extend to new protocols with reasonable assumptions. Second, the overhead of mixing multiple protocols should be minimized such that the overhead is not a performance bottleneck in any scenario. A robust design should ensure that in any case the mixed execution does not perform worse than any single candidate protocol involved. Finally, as many transactional databases back user-facing applications, dynamically switching protocols online in response to workload changes is necessary to maintain performance.

While several recent studies focus on mixed concurrency control, they only address a part of the above challenges. For example, MOCC [28] and HSync [22] are designed to mix OCC and 2PL with minimal mixing overhead but fails to extend to other protocols; Callas [31] and Tebaldi [23] provide a general framework that can cover a large number of protocols, but their overhead of mixing candidate protocols is non-trivial in some scenarios and do not support online protocol switch thus failing to address the workload changes. Our prior work [25] envisions an adaptive database that mixes concurrency control, but does not includes a general framework to address these challenges.

In this paper, we present a general mixed concurrency control scheme **CormCC** (*Coordination-free and Reconfigurable Mixed Concurrency Control*) to systematically address all the aforementioned challenges. CormCC decomposes a database into partitions according to workload access patterns and assigns a specific protocol to each partition, such that a protocol can be used to process the parts of a workload they are optimized for. We then develop several criteria to regulate the mixed execution of multiple forms of concurrency control to maintain ACID properties. We show that under reasonable assumptions, this method allows correct mixed execution without coordination across different protocols, which minimizes the mixing overhead. In addition, we develop a general protocol switching method (i.e. reconfiguration) to support changing protocols online with multiple protocols running together; the key idea is to compose a *mediated* protocol compatible with both the old and new protocol such that switch process does not have to stop all transaction workers while minimizing the impact on throughput and latency. To validate the efficiency and effectiveness of CormCC, we develop a prototype main-memory database on multi-core systems that supports mixed execution and dynamic switching of three widely-used protocols: a single-threaded partition based concurrency control (*PartCC*) from H-Store [10], an optimistic concurrency control (*OCC*) based on Silo [27], and a two-phase locking based on VLL (*2PL*) [21]. [1]

The main contributions of this paper over our vision paper [25] are the following:

- A detailed analysis of mixed concurrency control design space, a general framework that is not limited to a specific set of protocols to mix multiple forms of concurrency control without introducing extra overhead of coordinating conflicts across protocols.

- A general protocol switching method to reconfigure a protocol for parts of a workload without stopping the system or introducing significant overhead.

- A thorough evaluation of state-of-the-art mixed concurrency control approaches, CormCC's end-to-end performance over varied workloads, and the performance benefits and overhead of CormCC's mixed execution and online reconfiguration.

2 Related Work

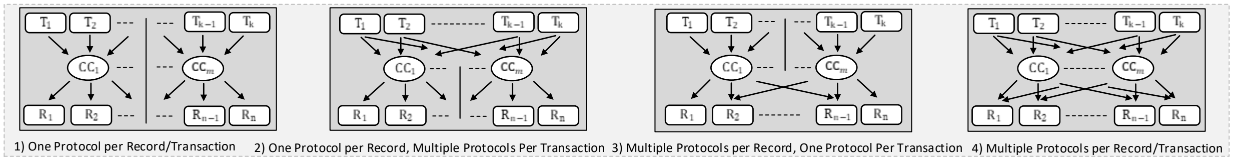
With the increase of main memory capacity and the number of cores for a single node, recent research focuses on improving traditional concurrency control protocols on modern hardware. We classify these works into optimizing single protocols, mixing multiple protocols, and adaptable concurrency control.

Optimizing Single Protocols Many recent projects consider optimizing a single protocol, which we believe are orthogonal to this project. H-Store [8, 10] developed a partitioned based concurrency control (PartCC) to minimize concurrency control overhead. The basic idea is to divide databases into disjoint partitions, where each partition is protected by a lock and managed by a single thread. A transaction can execute when it has acquired the locks of all partitions it needs to access. Exploiting data partitioning and the single-threaded model is also adopted by several research systems [8, 10, 32, 18, 19, 11]. Other projects propose new concurrency controls optimized for multi-core main-memory databases [6, 14, 27, 17, 35, 20, 13, 29, 30, 36, 12, 37, 16] or new data structures to remove the bottlenecks of traditional concurrency control [34, 21, 15, 9].

Mixing Multiple Protocols Callas [31] and its successive work Tebaldi [23] provide a modular concurrency control mechanism to group stored procedures and provide an optimized concurrency control protocol for each group based on offline workload analysis; for transaction conflicts across groups, Callas and Tebaldi introduce one or multiple protocols in addition to protocols for each group to resolve them. While stored procedure oriented protocol assignment can process conflicts within the same group more efficiently, the additional concurrency control overhead from executing both in-group and cross-group protocols can become the performance bottleneck for a main-memory database on a multi-core server. Section 3 shows a detailed discussion. In addition, the grouping based on offline analysis assumes workload conflicts are known upfront, which may not be true in real applications. Our work differs from Callas and Tebaldi in that our mixed concurrent control execution does not introduce any extra concurrency control overhead, which greatly reduces the mixing overhead. Additionally, we do not assume a static workload or require any knowledge about workload conflicts in advance, but allow protocols are chosen and reconfigured online in response to dynamic workloads. Some other projects exploit the mixed execution of 2PL or OCC [33, 6, 28, 22]. CormCC differs in that our framework is more general and can be extended to more protocols.

Adaptable Concurrency Control Adaptable concurrency control has been studied in several research works.

[1] Note that we use strong 2PL (SS2PL), but refer to it as 2PL



1) One Protocol per Record/Transaction 2) One Protocol per Record, Multiple Protocols Per Transaction 3) Multiple Protocols per Record, One Protocol Per Transaction 4) Multiple Protocols per Record/Transaction

Figure 1: Design choices of mixed concurrency control

At the hardware level, ProteusTM [7] is proposed to adaptively switch across multiple transaction memory algorithms for different workloads, but cannot mix them to process different parts of a workload. Tai et al. [24] shows the benefits of adaptively switching between OCC and 2PL. RAID [4, 3] proposes a general way to change a single protocol online. CormCC differs in that it allows multiple protocols running in the same system and supports to reconfigure a protocol for parts of workload. In this scenario, the presence of multiple protocols during the reconfiguration presents new challenges we are addressing.

3 Design Choices

While combining multiple protocols into a single system can potentially allow more concurrency to improve the overall throughput, it comes with the cost of higher concurrency control overhead. In this section, we discuss two key design choices to explore this trade-off and show how this trade-off motivates the design of CormCC. Specifically, we consider whether a record can be accessed (i.e. read/write) by multiple protocols and whether a single transaction involves multiple protocols. Based on the design choices, the overhead can be defined as *the cost of executing more than one protocol for each transaction plus the cost of synchronizing the concurrent read/write operations across different protocols for each record in the database*. Note that in this paper, we assume all the transaction logic and the corresponding concurrency control logic of a transaction are executed by a single thread (i.e. transaction worker), which is a common model in the design of mixed concurrency control [31, 23, 28, 22, 6]. We now discuss the four possible designs that are shown in Figure 1.

One Protocol per Record and per Transaction This is the simplest case, which is the left most part of Figure 1. We see that each transaction (denoted by `T`) can only choose one protocol (i.e. `CC` in Figure 1) and each record (denoted by `R`) can be accessed via one protocol. While the mixing overhead is minimal (based on our overhead definition), this design has very limited applicability since each transaction can only access the partition of records managed by a specific protocol. To the best of our knowledge, no previous work adopts it.

One Protocol per Record, Multiple Protocols per Transaction This design further allows that one transaction executes multiple protocols (shown in the second design in Figure 1); it provides the flexibility that transactions can access any record and allows a specific protocol to process all access to each record according to their access patterns. On the other hand, the execution of a single transaction may involve a larger set of instructions from multiple protocols, which makes CPU instruction cache less efficient. However, according to our experiment in Section 6.5 this mixing overhead is very low. MOCC [28] adopts this design to mix OCC and 2PL, but does not have a general framework.

Multiple Protocols per Record, One Protocol per Transaction An alternative design (the third one in Figure 1) is that each record can be accessed via multiple protocols and each transaction executes one protocol. This design is useful when the semantics of a subset of transactions (e.g. from stored procedures) can be leveraged by an optimized protocol. It raises a problem, however, that co-existence of multiple protocols on the same set of records should be carefully synchronized. One solution is to let all protocols share the same set of concurrency control metadata and carefully design each protocol such that all concurrent access to the same records can be synchronized without introducing additional coordination. This solution requires specialized design for all protocols and thus is limited in its applicability. Prior works Hekaton [6] and HSync [22] adopt this design, but can only combine 2PL and OCC.

Multiple Protocols per Record and per Transaction This design (the fourth one in Figure 1) provides the most fine-grained and flexible mixed concurrency control; each transaction can mix multiple protocols and each record can be accessed via different protocols. To process the concurrent access from different protocols over the same records, additional protocols are introduced. For example, Callas [31] and its successive work Tebaldi [23] organized protocols into a tree, where the protocols in leaf nodes process conflicts of the assigned transactions and the protocols in interior nodes process conflicts across its children. The overhead here is that for each record access, multiple concurrency control logic should be executed to resolve the conflicts across different protocols over that record. Such overhead, as we show in Section 6.3, can become a performance bottleneck in the multi-core main-memory database.

Summary The above discussion (and our experiments) show that the second design, which lets each pro-

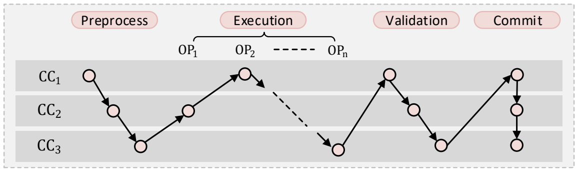


Figure 2: An Example of CormCC execution

tocol exclusively process the access of a subset of records to minimize synchronization cost and allows protocols are mixed within a transaction to provide mixing flexibility, strikes a good trade-off between leveraging the performance benefits of single protocols and minimizing mixing overhead. CormCC draws its spirit and builds a general and coordination-free framework.

4 CormCC Design

We consider a main-memory database with multiple forms of concurrency control on a multi-core machine. Each table includes one primary index and zero or more secondary indices. A transaction can be composed into read, write (i.e. update), delete, and insert operations to access the database via either primary or secondary indices in a key-value way. The database is (logically) partitioned with respect to candidate protocols within the system, that is, each partition is assigned a single protocol. Each protocol maintains an independent set of metadata for all records. For all operations on the records of a partition, the associated protocol executes its own concurrency control logic to process these operations (e.g. preprocess, reading/writing, commit), which we denote as *a protocol managing this partition*. We use a concurrency control lookup table to store the mapping from primary keys to the protocol. The lookup table maintains the mapping for the whole key space and is shared by all transaction workers. Prior work [26] shows that such a lookup table can be implemented in a memory-efficient and fast way, and thus will not be the performance bottleneck of CormCC. CormCC regards secondary indices as logically additional tables storing entries to primary keys (not pointers to records); it adopts a dedicated protocol (e.g. OCC) to process all concurrent operations over secondary indices. Transactions are routed to a global pool of transaction workers, each of which is a thread or a process occupying one physical core. This worker executes the transaction to the end (i.e. commit or abort) without interruption. We additionally use a coordinator to manage the online protocol reconfiguration for all transaction workers. It collects statistical information periodically from all workers, builds a new lookup table accordingly, and finally lets all workers use it. We first outline the CormCC protocol and then show the correctness of CormCC. After that, we discuss online protocol reconfiguration within CormCC.

4.1 CormCC Protocol

CormCC divides a transaction's life cycle into four phases: Preprocess, Execute transaction logic (Execution), Validation, and Commit. We adopt this four-phase model because most concurrency control protocols can fit into it. We use a transaction execution example in Figure 2 to explain the four phases.

Preprocess The preprocess phase executes concurrency control logic that should be executed before the transaction logic. Figure 2 shows that CormCC iterates over all candidate protocols (denoted as CC) and executes their preprocess phases respectively. Typical preprocess phase includes initializing protocol-specific metadata. For example, 2PL Wait-die needs to acquire a transaction timestamp to determine the relative order to concurrent transactions, or partition-based single-thread protocol (e.g. PartCC) acquires locks in a predefined order for partitions the transaction needs to access.

Execution Transaction logic is executed in this phase. As shown in Figure 2, for each operation (denoted as OP) issued from the transaction, CormCC first finds the protocol managing the record using the concurrency control lookup table. Then, CormCC utilizes the protocol's concurrency control logic to process this operation. For example, if the transaction reads an attribute of a record managed by 2PL, it acquires its read lock and returns the attribute's value. Note that insert operations can find the corresponding protocol in the lookup table even though the record to be inserted is not in the database because the table stores the mapping of the whole key space.

Validation Each validation phase of all protocols are executed sequentially in this phase. If the transaction passes all validation, it enters the Commit phase; otherwise, CormCC aborts it. For example, if OCC is involved, CormCC executes its validation to verify whether records read by OCC during Execution have been modified by other transactions. If yes, the transaction is aborted; otherwise, it passes this validation.

Commit Finally, CormCC begins an atomic Commit phase by executing commit phases of all protocols. For example, one typical commit phase, like OCC, will apply all writes to the database and make them visible to other transactions via releasing all locks. Note that an abort can happen in the Execution or Validation phase, in which case CormCC calls the abort functions of all protocols respectively.

4.2 Correctness

We first show the criteria for CormCC to generate serializable schedule for a set of concurrent transactions (i.e. its result is equivalent to some serial history of these transactions) to guarantee the consistency of databases.

We then discuss how CormCC avoids deadlock, and show that CormCC is recoverable (i.e. any committed transaction has not read data written by an aborted transaction).

Serializability To guarantee that CormCC is serializable, we require all candidate protocols are commit ordering conflict serializable (COCSR). It means that if two transactions t_i and t_j have conflicts on a record r (i.e. t_i and t_j reads/writes r and at least one of them is a write) and t_j depends on t_i (i.e. t_i accesses r before t_j), then t_i must be committed earlier than t_j.

Given that all candidate protocols are COCSR, we now show that CormCC is also COCSR. Suppose that two transactions t_i and t_j have conflicts on a record set R, we consider two cases. First, if for any conflicted record $r \in R$ the conflicted operation of t_i accesses r before t_j, then we have t_j depends on t_i. Since any candidate protocol is COCSR, for each conflicted record r they ensure that t_i is committed before t_j. Therefore in this case, CormCC can maintain COCSR property. In the second case, there exist two records r_1 and $r_2 \in R$, and t_i accesses r_1 before t_j and t_j accesses r_2 before t_i. We have t_j depends on t_i and t_i depends on t_j. In this case, r_1 and r_2 must be managed by two separate protocols p_1 and p_2 since each single protocol is COCSR and it is not possible to form a conflict-cycle in one protocol. Consider p_1, which manages r_1. Because it is COCSR, it enforces that t_i is committed earlier than t_j since t_j depends on t_i based on their conflicts on r_1. On the other hand, p_2 enforces that t_i is committed earlier than t_j. Therefore, there is no valid commit time for both t_i and t_j to suffice the above two constraints. Thus, in this case committing both transactions is impossible and the COCSR property of CormCC is also maintained. Finally, since COCSR is a sufficient condition for conflict serializable [2], CormCC is conflict serializable.

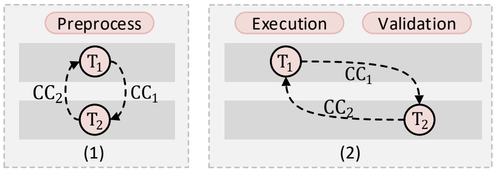


Figure 3: Examples of deadlock across protocols

Deadlock Avoidance While each individual protocol can provide mechanisms to avoid or detect deadlocks, mixing them using CormCC without extra regulation may not make the system deadlock-free. One potential solution can be using a global deadlock detection mechanism. However, this contradicts our spirit of not coordinating candidate protocols.

Alternatively, we examine the causes of deadlock and find that under reasonable assumptions, CormCC can mix single protocols without coordination. Specifically, the deadlock can happen within a single phase or across phases when two transactions wait for each other due to their conflicts on records that are managed by separate protocols. Figure 3 shows two such cases. The first case shows that the single-phase deadlock happens because both protocols CC_1 and CC_2 can make transactions T_1 and T_2 wait within one phase. For the second case, we see that T_1 waits for T_2 in the Execution phase due to CC_1 and additionally introduce conflicts to make T_2 wait for itself based on CC_2 in the Validation phase. Such deadlock is possible because a protocol (e.g. CC_2) can introduce conflicts across phases, that is, the conflicts introduced by T_1 in Execution are detected by T_2 in Validation.

CormCC avoids the two cases by requiring each protocol make transactions wait due to conflicts in no more than one phase, and in each phase only one protocol make transactions wait because of conflicts. If CormCC can meet the two criteria and each protocol can avoid or detect deadlocks, then CormCC is deadlock-free.

Recoverable To guarantee CormCC recoverable, we require that each candidate protocol is strict, which means that a record modified by a transaction is not visible to concurrent transactions until the transaction commits. This ensures transactions never read dirty writes (i.e. uncommited writes) and thus are recoverable [2]. Since each record written by a transaction is managed by a single strict protocol, the execution of CormCC will never read dirty writes and is recoverable.

Supported Protocols A candidate protocol incorporated in CormCC should be COCSR and strict. We find a wide range of protocols can meet these criteria including traditional 2PL and OCC [2], VLL [21], Orthrus [20], PartCC from H-Store [10], and Silo [27].

To enforce that CormCC meets the criteria of deadlock-free, CormCC requires each protocol specifies the phases where it makes transactions wait. Based on these specification, it is easy to detect whether the above criteria hold for a given set of protocols.

4.3 Online Reconfiguration

Online reconfiguration is to switch a protocol for a subset of records without stopping all transaction workers. CormCC uses a coordinator to manage this process. At first, the coordinator collects statistical information from all workers periodically and generates a new concurrency control lookup table. When a worker completes a transaction, it adopts the new lookup table while workers that have not finished their current transactions still use the old one. After all workers use the new table, the old one is deleted. We now introduce the challenge of supporting such online protocol switch.

Challenge The potential problem is that some transaction workers may use the new lookup table while others

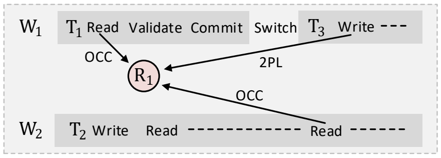


Figure 4: Problems during protocol reconfiguration

are using the old one. Figure 4 shows an example of this problem. Consider two workers W_1 and W_2, which execute transactions T_1 and T_2 respectively. Assume that both T_1 and T_2 access a record (i.e. R_1) that is managed by OCC. During their execution, the coordinator informs that the protocol managing R_1 should be switched to 2PL. Therefore, after T_1 finishes W_1 checks this message, performs the switch (i.e. using 2PL to access R_1), and starts a new transaction (T_3). Since OCC and 2PL maintain a different set of metadata, T_3 and T_2 are not aware of the conflict of T_2 reading R_1 and T_3 writing R_1, which may make database result in an inconsistent state.

Mediated Switching To address this issue, we propose mediated switching; it adopts a *mediated protocol* that is compatible with both old and new protocols. During reconfiguration, the coordinator lets all workers asynchronously change the old protocol to the mediated one; After all workers use the mediated protocol, the coordinator then informs them to adopt the new protocol.

We compose a mediated protocol that can execute concurrency control logic of both old and new protocols. Specifically, a mediated protocol first executes the Preprocess logic of both old and new protocols; then, it enters the Execution phase, where for each record access the mediated protocol executes the Execution logic of both protocols. The mediated protocol's Validation, Commit, and Abort also executes the corresponding logic of both protocols. While it is easy to compose a protocol that executes the logic of both protocols, one problem is how to unify different ways of applying modifications (i.e. insert/delete/write) of different protocols. We find there are two ways to apply modifications: in-place modification during execution and lazy modification during commit phase. In the mediated protocol, we always opt for lazy modification, which means storing the modification in a local buffer during execution and applying them when the transaction is committed. Since all protocols are strict, deferring the actual modification to the commit phase does not violate correctness of protocols. For example, 2PL performs in-place write, while OCC writes the new value into a local buffer and applies it in the commit phase. In our mediated protocol, we choose the OCC approach of applying writes.

We use the example in Figure 5 to illustrate this process, where we need to switch the protocol managing R_1 from OCC to 2PL. The mediated protocol here will execute the logic of both OCC and 2PL (denoted as OPCC). OPCC adopts the following logic:

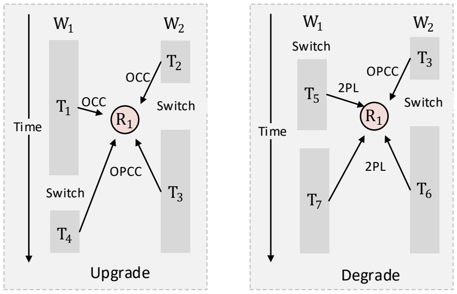


Figure 5: An Example of mediated switching

- If OPCC reads a record, it applies a read lock (2PL) and reads its value and timestamp into the read set (OCC).
- If OPCC writes a record, it applies a write lock (2PL) and stores the record along with new data into the write set (OCC).
- In the validation phase, it locks all records in write set (e.g. for critical section of Silo OCC)[2] and then validate the read set using OCC logic.
- In the commit phase, it applies all writes and release locks acquired by OCC and 2PL respectively.

The switch via mediated protocol is composed of two phases: *upgrade* and *degrade*. The protocol switch is initiated when the coordinator finds that the protocol for a record set *RS* should be changed. It starts the *upgrade phase* by issuing a message to all workers to let them switch the protocol for *RS* to the mediated protocol. Each transaction worker checks for this message between transactions and acknowledges the message to the coordinator. During this asynchronous process, workers that have received the message access *RS* using the mediated protocol, while other workers may access *RS* using the old protocol, which happens when they are running transactions that started before the switch. The left part of Figure 5 shows an example of upgrade phase. We see that worker W_2 finished T_2 first; thus, it begins to access R_1 using OPCC (i.e. in transaction T_3). At the same time, T_1 still accesses R_1 using OCC. According to the above OPCC description, we see that the conflicts on R_1 from W_1 and W_2 can be serialized because OPCC runs the full logic of OCC. After all workers acknowledge the switch for *RS*, the degrade phase begins with the coordinator messaging workers about the degrade to the new protocol. Therefore, workers are using either the mediated protocol or the new protocol, and serializability is guaranteed by the new protocol logic (e.g. 2PL in our example) used in both execution modes. The right part of Figure 5 shows an example of degrade phase, where the conflicts over R_1 can be serialized because OPCC also executes the full logic of 2PL.

[2] Note that OCC and 2PL use different sets of metadata; no deadlock can exist between them

5 Prototype Design

We build a prototype main-memory multi-core database that can dynamically mix PartCC from H-Store [10], OCC from Silo [27], and 2PL from VLL [21] using CormCC. PartCC partitions the database and associates each partition an exclusive lock. Every transaction first acquires all locks for the partitions it needs to read/write in a predefined order before the transaction logic is executed. Then, the transaction is executed by a single thread to the end without additional coordination. In Silo OCC, each record is assigned a timestamp. During transaction execution, Silo OCC tracks read/write operations, and stores the records read by the transactions along with their associated timestamps into a local read set and all writes into a local write set. In the validation phase, Silo OCC locks the write set and validates whether records in read set are changed using their timestamps. If the validation succeeds, it commits; otherwise, it aborts. VLL is an optimized 2PL by co-locating each lock with each record to remove the contention of the centralized lock manager.

Our prototype partitions primary indices and corresponding records using an existing partitioning algorithm [5]. Each partition is assigned with a transaction worker (i.e. thread or process) and each worker is only assigned transactions that will access some data in its partition (the *base partition*), but may also access data in other partitions (the *remote partition*). CormCC selects a protocol for each partition according to its access pattern. A partition routing table maintains the mapping from primary keys to partition numbers and also corresponding protocols. We use stored procedures as the primary interface, which is a set of predefined and parameterized SQL statements. Stored procedures can provide a quick mapping from database operations to the corresponding partitions by annotating the parameters that can be used to identify a base partition to execute the transaction and other involved partitions [10].

The mixed execution of the three protocols start with Preprocess phase, where PartCC acquires all partition locks in a predefined order. Transactions may wait in this phase because of partition lock requests. Next, transactions enter Execution, where CormCC uses PartCC, OCC, and 2PL to process record access operations according to which partition the record belongs to. Note that only 2PL will make transactions wait in this phase, so transactions in the Execution phase will not wait for those blocked in the Preprocess, which indicates deadlocks across PartCC and 2PL is impossible. After the Execution phase, Validation begins and OCC acquires write locks and validates the read set [27]. Only OCC requires transactions to wait; thus, they will not be blocked by previous phases. Finally, there is no wait in commit phase; all protocols apply writes and release all locks. We show that such mixed execution is correct. First, for PartCC and 2PL if a transaction t_i conflicts with another transaction t_j, t_i cannot proceed until t_j commits or vice versa, so PartCC and 2PL are COCSR. Shang et al. [22] have proven that Silo OCC is also COCSR. Therefore, their mixed execution using CormCC maintains COCSR. In addition, each of PartCC, 2PL, and OCC can independently either avoid or detect deadlocks and make transactions conflict-wait in only one mutually exclusive phase among Preprocess, Execution, or Validation. Thus, their mixed execution can also prevent or detect deadlocks. Finally, all protocols are strict, so is their mixed execution.

To enable dynamic protocol reconfiguration, we build two binary classifiers to predict the ideal protocol for each partition. The detailed discussion of classifiers are presented in a technical report [1].

6 Experiments

We now evaluate the effectiveness of mixed execution and online reconfiguration of CormCC. Our experiments answer four questions: 1) How does CormCC perform compared to state-of-the-art mixed concurrency control approaches (Section 6.3)? 2) How does CormCC adaptively mix protocols under varied workloads over time (Section 6.4)? 3) What is the performance benefit and overhead of mixed execution (Section 6.5)? 4) What is the performance benefit and overhead of online reconfiguration (Section 6.6)?

All experiments are run on a single server with four NUMA nodes, each of which has a 8-core Intel Xeon E7-4830 processor (2.13 GHz), 64 GB of DRAM and 24 MB of shared L3 cache, yielding 32 physical cores and 256 GB of DRAM in total. Each core has a private 32 KB of L1 cache and 256 KB of L2 cache. We disable hyperthreading such that each worker occupies a physical core. To eliminate network client latency, each worker combines a client transaction generator.

6.1 Prototype Implementation

We develop a prototype based on Doppel [17], an open-source multi-core main-memory transactional database. Clients issue transaction requests using pre-defined stored procedures, where all parameters are provided when a transaction begins, and transactions are executed to the completion without interacting with clients. Stored procedures issue read/write operations using interfaces provided by the prototype. Each transaction is dispatched to a worker that runs this transaction to the end (commit or abort).

Workers access records via key-value hash tables. Each worker thread occupies a physical core and main-

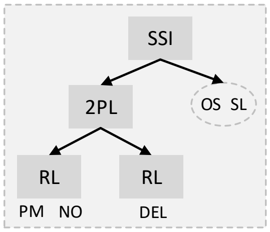


Figure 6: A three-layer configuration of Tebaldi

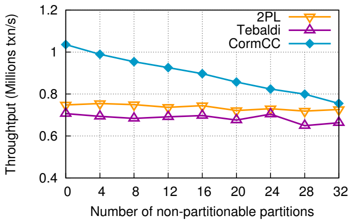


Figure 7: Comparison under different partitionability

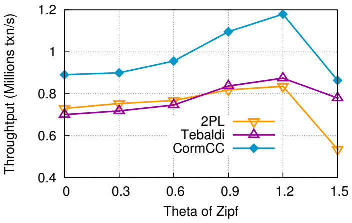


Figure 8: Comparison unber different conflicts

tains its own memory pool to avoid memory allocation contention across many cores [34]. A coordinator thread is used for extracting features for the prediction classifiers from statistics collected by workers and predicting the ideal protocol to be used. Our prototype supports automatically selecting PartCC [10], Silo OCC [27], or No-Wait VLL [21] for each partition. Note that we have compared 2PL variants No-Wait and Wait-Die, and find that 2PL No-Wait performs best in most cases because of lower synchronization overhead of lock management [20]. We do not implement logging in CormCC since prior work shows that logging is not a performance bottleneck [17, 38]. Our comparison additionally includes a general mixed concurrency control framework based on Tebaldi [23] (denoted as Tebaldi) and a hybrid approach of OCC and 2PL [22, 28] (denoted as Hybrid), that adopts locks to protect highly conflicted records but uses validation for the rest. We statically tune the set of highly conflicted records to make Hybrid have the highest throughput. Specifically, for our highly conflicted workload we protect 1000 mostly-conflicted records for each partition and for our lowly conflicted workload no records will be locked and we use OCC for them.

6.2 Benchmarks & Experiment Settings

We use YCSB and TPC-C in our experiments. We generate one table for YCSB that includes 10 million records, each with 25 columns and 20 bytes for each column. Transactions are composed of mixed read and read-modify-write operations. The partitioning of YCSB is based on hashing its primary keys. TPC-C simulates an order processing application. We generate 32 warehouses and partition the store according to warehouse IDs except the Item table, which is shared by all workers. We use the full mix of five procedures.

To generate varied workloads, we tune three parameters. The first is the percentage of cross-partition transactions ranging from 0 to 100. We set the number of partitions a cross-partition transaction will access as 2; The second is the mix of stored procedures for a workload. Note that YCSB only has one stored procedure, and we tune the number of operations per transaction and the ratio of read operations. Finally, we vary data access skewness. We use Zipf to generate record access distribution within a partition. *Theta* of Zipf can be varied from 0 to 1.5. This means for TPC-C within a partition determined by WarehouseID, we skew record access for related tables. We also vary these parameters to train our classifiers for protocol prediction. The detailed configuration of training classifiers is illustrated in [1].

6.3 Comparison with Tebaldi

Tebaldi [23] is a general mixed concurrency control framework that groups stored procedures according to their conflicts and build a hierarchical concurrency control protocols to address in-group and cross-group conflicts. Figure 6 shows a three-layer configuration for TPC-C. We see that NewOrder (NO) and Payment (PM) are in the same group and their conflicts are managed by runtime pipeline (RL). Runtime pipeline is an optimized 2PL that can leverage the semantics of stored procedures to pipeline conflicted transactions. Delivery (DEL) alone is in another group also run by runtime pipeline. Conflicts between the two groups are processed by 2PL. OrderStatus (OS) and StockLevel (SL) are executed in a separate group. Their conflicts with the rest of the groups are processed by Serializable Snapshot Isolation (SSI). For every operation issued by a transaction, it needs to execute all concurrency control logic from the root node to the leaf node to delegate the conflicts to specific protocols. For example, any operation issued by NewOrder needs to go through the logic of SSI, 2PL, and RL. While this approach can process conflicts in a more fine-grained way, the overhead of multiple protocols for all operations can limit the performance. Another restriction of Tebaldi is that it relies on the semantics of stored procedures to assign protocols. For workloads including data dependent behaviour (e.g. hot keys or affinity between keys) and not having stored procedures with rich semantics (i.e. YCSB in our test), Tebaldi can only use one protocol to process the whole workload. In our test, we use this 3-layer configuration for Tebaldi, which has the best performance for TPC-C [23]. Note that we use an optimized Runtime Pipeline reported in [31], which can eliminate the conflicts between Payment and NewOrder.

We first compare CormCC with Tebaldi, implemented in our prototype, using TPC-C over mixing well-partitionable and non-partitionable workloads. We par-

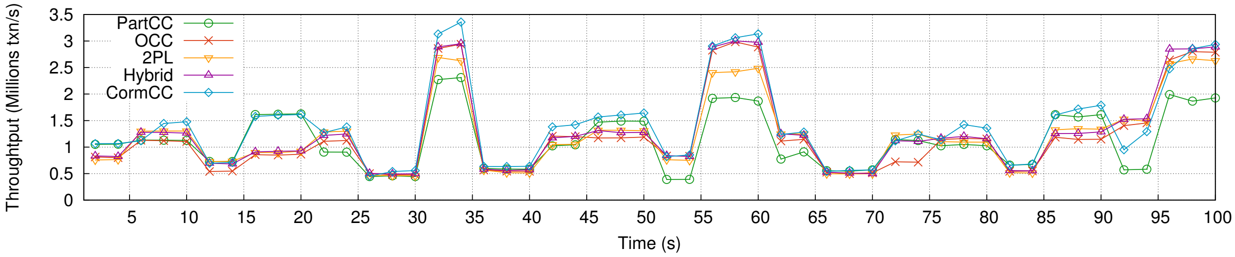


Figure 9: Holistic test for CormCC under YCSB varied workloads over time

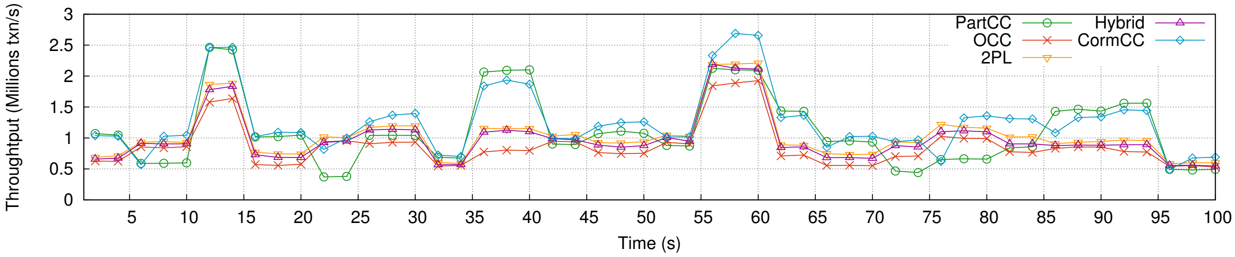


Figure 10: Holistic test for CormCC under TPC-C varied workloads over time

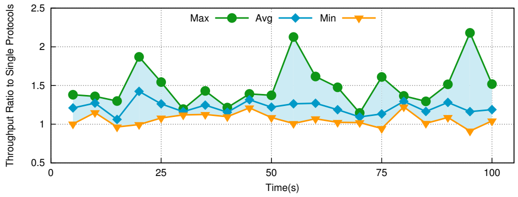


Figure 11: CormCC throughput ratio to single protocols for YCSB

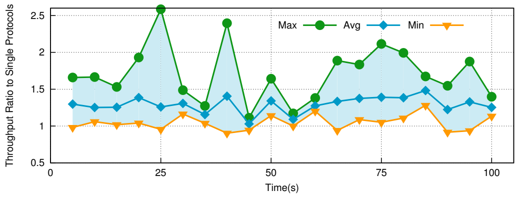


Figure 12: CormCC throughput ratio to single protocols for TPC-C

tition the database into 32 warehouses and start our test with a well-partitionable workload (i.e. each partition receives 100% single-partition transactions), and then increase the number of non-partitionable warehouses (i.e. each receives 100% cross-partition transactions) by an interval of 4. Throughout this test, we use default transaction mix of TPC-C. Since both Tebaldi and CormCC use 2PL as their candidate protocol, our test additionally includes the results of 2PL.

Figure 7 shows the performance results of three protocols. CormCC first adopts PartCC and then proceeds to mix PartCC and 2PL for workloads with mixed partitionability. When the workload becomes non-partitionable, 2PL is used by CormCC. We see that CormCC always performs better than Tabaldi and 2PL because it can leverage the partitionable workloads. Tebaldi always performs slightly worse than 2PL because of its concurrency control overhead from multiple protocols. While such overhead is not substantial in a distributed environment as shown in the original paper [23], it can become a bottleneck in a main-memory multi-core database due to the elimination of network I/O operations.

To highlight Tebaldi's performance benefits of efficiently processing conflicts, we increase the access skewness within each warehouse by varying the *theta* of Zipf distribution from 0 to 1.5 with an interval 0.3. Here, we choose 16 warehouses as partitionable and the rest as non-partitionable. Figure 8 shows that the throughput of all protocols increases at first, because more access skewness introduces better access locality and improves CPU cache efficiency. Then, high conflicts dominate the performance and the throughput decreases for all protocols. We see that with higher conflicts Tebaldi gradually outperforms 2PL, and suffers less throughput loss in the workload with very high conflicts.

These tests show that while Tebaldi can efficiently process conflicts, it comes with a non-trivial concurrency control overhead. In addition, Tebaldi needs to know the conflicts of a workload a priori such that it can utilize the static analysis [23] to make an efficient configuration offline. In contrast, CormCC mixes protocols with minimal overhead, requires no knowledge of conflicts beforehand, and can dynamically choose protocols online.

6.4 Tests on Varied Workloads

We evaluate the holistic benefits of CormCC by running YCSB with randomizing benchmark parameters every 5 seconds. We compare the same randomized run (e.g. same parameters at each interval) with fixed protocols and Hybrid. CormCC collects features every second and concurrently selects the ideal protocol for each partition.

We randomly vary five parameters: i) read rate chosen in 50%, 80%, and 100%; ii) number of operations per transaction, selected between 15 and 25; iii) theta of Zipf for data access distribution; chosen from 0, 0.5, 1, and

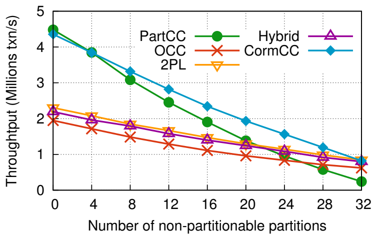


Figure 13: Measuring partitionability

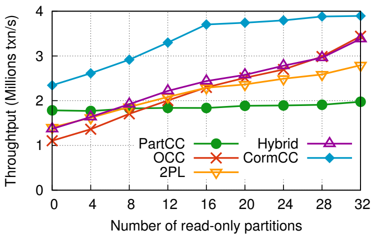


Figure 14: Measuring read/write ratio

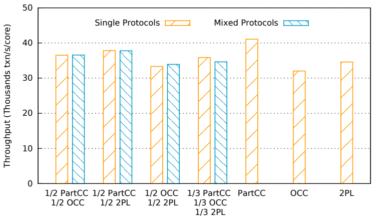


Figure 15: Testing the overhead of mixed execution

1.5; iv) the number of partitions that have cross-partition transactions (with the remaining partitions as well partitionable): we randomly choose the number from 0 to 32 with the interval 4; v) the percentage of cross-partition transactions for partitions in (iv), randomly selected between 50% and 100%.

The test starts with a well-partitionable workload of 80% read rate, 15 operations per transaction, and a uniform access distribution (i.e. $theta = 0$). Figure 9 shows the test results of every 2 seconds for 100 seconds in total. We see that in almost all cases CormCC can either choose the best protocol or find a mixed execution to outperform any candidate protocols and Hybrid approach, while not experiencing long periods of throughput degradation due to switching. CormCC can achieve at most 2.5x, 1.9x, 1.8x, and 1.7x throughput of PartCC, OCC, 2PL, and Hybrid respectively.

We additionally test the performance variations of CormCC under randomized varied workloads of TPC-C. We partition the database into 32 warehouses, and have each worker collect features every second and select the ideal protocol for each warehouse at runtime. We permute four parameters to generate varied workloads. First, we randomly select a transaction mix in 10 candidates, where one is default transaction mix of TPC-C and other nine are randomly generated. Then, we vary the three parameters: record skew for related tables, the number of warehouses that have cross-partition transactions, and the percentage of cross-partition transactions in the same way as YCSB test. We report the results of every 2 seconds in Figure 10. The test starts with well-partitionable default transaction mix of TPC-C and varies workload every 5 seconds. We see that it has similar behaviours of Figure 9, where CormCC can almost always perform the best. In this test, CormCC can achieve at most 2.8x, 2.4x, 1.7x, and 1.8x throughput of PartCC, OCC, 2PL, and Hybrid respectively.

We then report the ratios of the mean throughput of CormCC (after protocol switching) to that of the worst and best single protocols (labeled by `max` and `min` respectively) for each varied workload (i.e. every 5s) of both benchmarks in Figure 11 and Figure 12. We additionally report the ratio of the mean throughput of CormCC to the average throughput of three single fixed protocols (labelled by `avg`) in each varied workload. We see that the highest ratio CormCC can achieve for YCSB and TPC-C is 2.2x and 2.6x respectively. For 55% workloads of YCSB and 85% workloads of TPC-C, the average ratio is at least 1.2x. The lowest ratio in YCSB and TPC-C test is 0.91x and 0.94x respectively, which means that for the two benchmarks CormCC can achieve at least 91% and 94% throughput of the best protocol due to wrong protocol selection for some partitions. These results show that CormCC can achieve significant performance gains when a wrong protocol is selected for a workload, can improve the throughput over single protocols for a wide range of varied workloads, and is robust to dynamic workloads.

6.5 Evaluating Mixed Execution

In this subsection, we first evaluate the performance benefits of CormCC over single protocols and Hybrid approaches, and then test the overhead of CormCC.

We first show how mixed well-partitionable and non-partitionable workloads based on YCSB benchmark influence the relative performance of CormCC to other protocols. We partition the database into 32 partitions, and start our test with a well-partitionable workload and then increase the number of non-partitionable partitions by an interval of 4. In this test, each transaction includes 80% read operations. For these tests, we use transactions consisting 20 operations and skew record access within each partition using Zipf distribution with $theta = 1.5$.

Figure 13 shows that CormCC always performs best because it starts with PartCC and then adaptively mixes PartCC for partitionable workloads and 2PL for highly conflicted non-partitionable counterpart. Compared to CormCC, the performance of PartCC degrades rapidly due to high partition conflicts and other protocols cannot take advantage of partitionable workloads.

We then test workloads with the increasing percentage of read operations. Initially, operations accessing each partition include 80% read operations; we increase the number of partitions receiving 100% read operations by an interval of 4. In this test, our workload includes 16 partitions having 100% cross-partition transactions among them, while the others are only accessed by single-partition transactions.

Figure 14 shows that CormCC has remarkable throughput improvement over other protocols by combining the benefits of PartCC, OCC, and 2PL. Specifically, CormCC first mixes PartCC and 2PL, and then applies OCC for non-partitionable and read-only partitions. While Hybrid can adaptively mix OCC and 2PL, it is sub-optimal due to failing to leverage the benefits of PartCC. In these tests, the speed-ups of CormCC over PartCC, OCC, 2PL, and Hybrid can be up to 3.4x, 2.2x, 1.9x, and 2.0x respectively.

Next, we test the overhead of CormCC. We first execute transactions using CormCC and track the percentage of each transaction's operations executed on records owned by a specific protocol (e.g. 1/2 of the transaction's records use OCC and 1/2 of the transaction's records use 2PL). With the percentages collected, we execute a mix of transactions where a corresponding percentage of the transactions are executed exclusively on a single protocol (e.g. 1/2 of the transactions are only OCC and 1/2 are only 2PL), and compare the throughputs of the two approaches. Note that to test the overhead without involving the performance advantages of CormCC over single protocols, we use a single core to execute all transactions.

Figure 15 shows a micro-benchmark to evaluate mixed execution overhead. We execute 50,000 transactions, each having 20 operations with 50% read operations, with the rest as read-modify-write operations. Key access distribution is uniform. The dataset is partitioned into 32 partitions; 10 of them are managed by OCC, 10 of them are for 2PL, and 12 are PartCC. The "mixed protocols" shows the average throughput of CormCC with different percentage of operations executed by different protocols; the "single protocol" results show the average throughput of a single protocol (e.g. 100% of transactions use OCC), or using single protocols to exclusively execute a corresponding percentage of transactions. We find that our method has roughly the same throughput as a mix of "single protocols", which shows that the overhead of mixed concurrency control is minimal in CormCC. This is largely due to the fact that we do not add extra meta-data operations to synchronize conflicts across protocols.

6.6 Evaluating Mediated Switching

To evaluate the performance benefits and overhead of mediated switching (denoted as Mediated), we compare it with a method of stopping all protocol execution and applying the new protocol (denoted as StopAll). In our test, we perform a protocol switch from OCC to 2PL using five YCSB workloads with uniform key access distribution. The first workload only includes short-lived transactions with each having 10 read and 10 read-modify-write operations. The other workloads include a mix of short and long-running transactions. We generate long-running transactions by introducing client think/wait time to short transactions. The long transactions last 0.5s, 1s, 2s, and 4s for the four workloads respectively and are dedicated to one worker. We collect throughput every second and report the average throughput during protocol switching. We ensure that switch happens at the start, end, and middle of a long running transaction, which represent that switch waits for little, whole, and half of the transaction respectively, and report three test cases for each mixed workload.

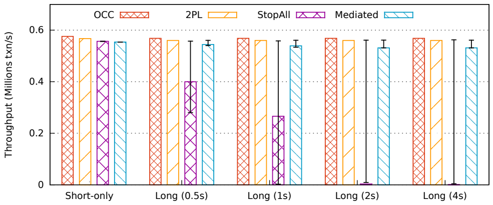


Figure 16: Testing mediated switching

As shown in Figure 16, we see that Mediated and StopAll have a minimal throughput drop compared to 2PL when the workload only includes short transactions. When long-running transactions are introduced, StopAll suffers due to waiting for the completion of long-running transactions, while Mediated can still maintain high throughput during the switch because Mediated does not stop all workers, but let them adopt both 2PL and OCC (i.e. upgrade phase); then, the coordinator notifies all workers to adopt 2PL (i.e. degrade phase) after the long transaction ends. Mediated protocol can achieve at least 93% throughput of OCC or 2PL due to the overhead of executing the logic of two protocols.

In addition, we perform the same test for all other pairwise protocol switching. We find that the overhead is minimal under short-only workload. When long-running transactions are introduced, the maximum throughput drop is about 20% during protocol switching from PartCC to OCC. This is acceptable compared to StopAll, which cannot process new transactions in the switch process. These experiments show that Mediated can maintain reasonable throughput during a protocol switch, even in the presence of long transactions.

7 Conclusion

By exploring the design space of mixed concurrency control, CormCC presents a new approach to generally mixing multiple concurrency control protocols, while not introducing coordination overhead. In addition, CormCC proposes a novel way to reconfigure a protocol for parts of a workload online with multiple protocols running. Our experiments show that CormCC can greatly outperform static protocols, and state-of-the-art mixed concurrency control approaches in various workloads.

References

[1] Toward Coordination-free and Reconfigurable Mixed Concurrency Control (Technical Report). `https://newtraell.cs.uchicago.edu/files/tr_authentic/TR-2018-06.pdf`.

[2] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing, 1986.

[3] BHARGAVA, B. K., HELAL, A., FRIESEN, K., AND RIEDL, J. Adaptility experiments in the RAID distributed data base system. In *Ninth Symposium on Reliable Distributed Systems, SRDS 1990, Huntsville, Alabama, USA, October 9-11, 1990, Proceedings* (1990), pp. 76–85.

[4] BHARGAVA, B. K., AND RIEDL, J. A model for adaptable systems for transaction processing. *IEEE Trans. Knowl. Data Eng. 1*, 4 (1989), 433–449.

[5] CURINO, C., ZHANG, Y., JONES, E. P. C., AND MADDEN, S. Schism: a workload-driven approach to database replication and partitioning. *PVLDB 3*, 1 (2010), 48–57.

[6] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), pp. 1243–1254.

[7] DIDONA, D., DIEGUES, N., KERMARREC, A., GUERRAOUI, R., NEVES, R., AND ROMANO, P. Proteustm: Abstraction meets performance in transactional memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016* (2016), pp. 757–771.

[8] HARIZOPOULOS, S., ABADI, D. J., MADDEN, S., AND STONEBRAKER, M. OLTP through the looking glass, and what we found there. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008* (2008), pp. 981–992.

[9] JUNG, H., HAN, H., FEKETE, A. D., HEISER, G., AND YEOM, H. Y. A scalable lock manager for multicores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), pp. 73–84.

[10] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S. B., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB 1*, 2 (2008), 1496–1499.

[11] KEMPER, A., AND NEUMANN, T. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany* (2011), pp. 195–206.

[12] KIM, K., WANG, T., JOHNSON, R., AND PANDIS, I. ERMIA: fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 1675–1687.

[13] KIMURA, H. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (2015), pp. 691–706.

[14] LARSON, P., BLANAS, S., DIACONU, C., FREEDMAN, C., PATEL, J. M., AND ZWILLING, M. High-performance concurrency control mechanisms for main-memory databases. *PVLDB 5*, 4 (2011), 298–309.

[15] LEVANDOSKI, J. J., LOMET, D. B., AND SENGUPTA, S. The bw-tree: A b-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013* (2013), pp. 302–313.

[16] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* (2014), pp. 479–494.

[17] NARULA, N., CUTLER, C., KOHLER, E., AND MORRIS, R. Phase reconciliation for contended in-memory transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* (2014), pp. 511–524.

[18] PANDIS, I., JOHNSON, R., HARDAVELLAS, N., AND AILAMAKI, A. Data-oriented transaction execution. *PVLDB 3*, 1 (2010), 928–939.

[19] PANDIS, I., TÖZÜN, P., JOHNSON, R., AND AILAMAKI, A. PLP: page latch-free shared-everything OLTP. *PVLDB 4*, 10 (2011), 610–621.

[20] REN, K., FALEIRO, J. M., AND ABADI, D. J. Design principles for scaling multi-core OLTP under high contention. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 1583–1598.

[21] REN, K., THOMSON, A., AND ABADI, D. J. Lightweight locking for main memory database systems. vol. 6, pp. 145–156.

[22] SHANG, Z., LI, F., YU, J. X., ZHANG, Z., AND CHENG, H. Graph analytics through fine-grained parallelism. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 463–478.

[23] SU, C., CROOKS, N., DING, C., ALVISI, L., AND XIE, C. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017* (2017), pp. 283–297.

[24] TAI, A. T., AND MEYER, J. F. Performability management in distributed database systems: An adaptive concurrency control protocol. In *MASCOTS '96, Proceedings of the Fourth International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems, February 1-3, 1996, San Jose, California, USA* (1996), pp. 212–216.

[25] TANG, D., JIANG, H., AND ELMORE, A. J. Adaptive concurrency control: Despite the looking glass, one concurrency control does not fit all. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings* (2017).

[26] TATAROWICZ, A., CURINO, C., JONES, E. P. C., AND MADDEN, S. Lookup tables: Fine-grained partitioning for distributed databases. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012* (2012), pp. 102–113.

[27] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 18–32.

[28] WANG, T., AND KIMURA, H. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB 10*, 2 (2016), 49–60.

[29] WANG, Z., MU, S., CUI, Y., YI, H., CHEN, H., AND LI, J. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 1643–1658.

[30] WU, Y., CHAN, C. Y., AND TAN, K. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 1689–1704.

[31] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRITSOS, M., AND WANG, Y. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), pp. 279–294.

[32] YAO, C., AGRAWAL, D., CHEN, G., LIN, Q., OOI, B. C., WONG, W., AND ZHANG, M. Exploiting single-threaded model in multi-core inmemory systems. *IEEE Trans. Knowl. Data Eng. 28*, 10 (2016), 2635–2650.

[33] YU, P. S., AND DIAS, D. M. Analysis of hybrid concurrency control schemes for a high data contention environment. *IEEE Trans. Software Eng. 18*, 2 (1992), 118–129.

[34] YU, X., BEZERRA, G., PAVLO, A., DEVADAS, S., AND STONEBRAKER, M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB 8*, 3 (2014), 209–220.

[35] YU, X., PAVLO, A., SANCHEZ, D., AND DEVADAS, S. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 1629–1642.

[36] Yuan, Y., Wang, K., Lee, R., Ding, X., Xing, J., Blanas, S., and Zhang, X. BCC: reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *PVLDB 9*, 6 (2016), 504–515.

[37] Zhang, Y., Power, R., Zhou, S., Sovran, Y., Aguilera, M. K., and Li, J. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 276–291.

[38] Zheng, W., Tu, S., Kohler, E., and Liskov, B. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* (2014), pp. 465–477.

Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow

John Sonchack[⋆]*, Oliver Michel*[†]*, Adam J. Aviv*[‡]*, Eric Keller*[†]*, and Jonathan M. Smith*[⋆]
[⋆]University of Pennsylvania, [‡]United States Naval Academy, and [†]University of Colorado, Boulder

Abstract

Measurement plays a key role in network operation and management. An important but unaddressed practical requirement in high speed networks is supporting concurrent applications with diverse and potentially dynamic measurement objectives. We introduce *Flow, a switch accelerated telemetry system for efficient, concurrent, and dynamic measurement. The design insight is to carefully partition processing between switch ASICs and application software. In *Flow, the switch ASIC implements a pipeline that exports telemetry data in a flexible format that allows applications to efficiently compute many different statistics. Applications can operate concurrently and dynamically on identical streams without impacting each other. We implement *Flow as a line rate P4 program for a 3.2 Tb/s commodity switch and evaluate it with four example monitoring applications. The applications can operate concurrently and dynamically, while scaling to measure terabit rate traffic with a single commodity server.

1 Introduction

Measurement plays a critical role in networking. Monitoring systems measure traffic for security [6, 35, 51, 36], load balancing [1, 26] and traffic engineering [55, 23, 27]; while engineers measure traffic and data plane performance to diagnose problems [14, 59, 25, 58, 56] and design new network architectures and systems [47, 5].

In high speed networks, which in 2018 have 100 Gb/s links and multi-Tb/s switches, it is challenging to support measurement without compromising on important practical requirements. Traditional switch hardware is inflexible and supports only coarse grained statistics [21, 45], while servers are prohibitively expensive to scale [50].

Fortunately, advances in switch hardware are presenting new opportunities. As the chip space and power cost of programmability drops [49, 7], switches are quickly moving towards reconfigurable ASICs [42, 44] that are capable of custom packet processing at high line rates. Recent telemetry systems [50, 40] have shown that these programmable forwarding engines (PFEs) can implement custom streaming measurement queries for fine-grained traffic and network performance statistics.

An open question, however, is whether telemetry systems can harness the flexibility and performance of PFEs *while also meeting requirements for practical deployment.* Current PFE accelerated telemetry systems [50, 40] focus on efficiency, compiling queries to minimize workload on servers in the telemetry infrastructure. Efficiency matters, but compiled queries do not address two other practical requirements that are equally important: concurrent measurement and dynamic queries.

First, support for concurrent measurement. In practice, there are likely to be multiple applications measuring the network concurrently, with queries for different statistics. A practical telemetry system needs to multiplex the PFE across all the simultaneously active queries. This is a challenge with compiled queries. Each query requires different computation that, given the line-rate processing model of a PFE [49], must map to dedicated computational resources, which are limited in PFEs.

Equally important for practical deployment is support for *dynamic* querying. As network conditions change, applications and operators will introduce or modify queries. A practical telemetry system needs to support these dynamics at runtime without disrupting the network. This is challenging with compiled PFE queries because recompiling and reloading the PFE is highly disruptive. Adding or removing a query pauses not only measurement, but also *forwarding* for multiple seconds.

Introducing *Flow. We introduce *Flow, a practical PFE accelerated telemetry system that is not only flexible and efficient, but also supports concurrent measurement and dynamic queries. Our core insight is that concurrency and disruption challenges are caused by compiling *too much* of the measurement query to the PFE, and can be resolved without significant impact to performance by carefully lifting parts of it up to software.

At a high level, a query can be decomposed into three logical operations: a *select* operation that determines which packet header and metadata features to capture; a *grouping* operation that describes how to map packets to flows; and a *aggregation function* that defines how to compute statistics over the streams of grouped packet features. The primary benefit of using the PFE lies in its capability to implement the select and grouping operations efficiently because it has direct access to packet headers and low latency SRAM [40]. The challenge is implementing aggregation functions in the PFE, which are computationally complex and query dependent.

`*Flow` is based on the observation that for servers, the situation is exactly reversed. A server cannot efficiently access the headers of every packet in a network, and high memory latency makes it expensive to group packets. However, once the packet features are extracted and grouped, a server can perform coarser grained grouping and mathematical computation very efficiently.

`*Flow`'s design, depicted in Figure 2, plays to the strengths of both PFEs and servers. Instead of compiling entire queries to the PFE, `*Flow` places parts of the *select* and *grouping* logic that are common to all queries into a match+action pipeline in the PFE. The pipeline operates at line rate and exports a stream of records that software can compute a diverse range of custom streaming statistics from *without needing to group per-packet records*. This design maintains the efficiency benefits of using a PFE while eliminating the root causes of concurrency and disruption issues. Further, it *increases* flexibility by enabling more complex aggregation functions than a PFE can support.

Grouped Packet Vectors. To lift the aggregation function off of the PFE, `*Flow` introduces a new record format for telemetry data. In `*Flow`, PFEs export a stream of *grouped packet vectors* (GPVs) to software processors. A GPV contains a flow key, e.g., IP 5-tuple, and a variable-length list of packet feature tuples, e.g., timestamps and sizes, from a sequence of packets in that flow.

Each application can efficiently measure different aggregate statistics from the packet feature tuples in the same GPV stream. Applications can also dynamically change measurement without impacting the network, similar to what a stream of raw packet headers [25] would allow, but without the cost of cloning each packet to a server or grouping in software.

Dynamic in-PFE Cache. Switches generate GPVs at line rate by compiling the `*Flow cache` to their PFEs, alongside other forwarding logic. The cache is an append only data structure that maps packets to GPVs and evicts them to software as needed.

To utilize limited PFE memory, e.g., around 10MB as efficiently as possible, we introduce a key-value cache that supports dynamic memory allocation and can be implemented as a sequence of match+action tables for PFEs. It builds on recent match+action implementations of fixed width key-value caches [40, 29, 50] by introducing a line rate memory pool to support variable sized entries. Ultimately, dynamic memory allocation increases the average number of packet feature tuples that accumulate in a GPV before it needs to be evicted, which lowers the rate of processing that software must support.

Implementation and Evaluation. We implemented the `*Flow` cache [1] for a 100BF-32X switch, a 3.2 Tb/s switch with a Barefoot Tofino [42] PFE that is programmable with P4 [8]. The cache is compiler-guaranteed to run at line rate and uses a fixed amount of hardware resources regardless of the number or form of measurement queries.

To demonstrate the practicality of `*Flow`, we implemented three example monitoring applications that measure traffic in different ways: a host profiler that collects packet level timing details; a traffic classifier that measures complex flow statistics; and a micro-burst attributer that analyzes per-packet queue depths. Although these applications measure different statistics, they all can operate concurrently on the same stream of GPVs and dynamically change measurements without disrupting the network. Benchmarks show that the applications can scale to process GPVs at rates corresponding to Terabit-rate traffic while using under 10 cores.

To further demonstrate the practicality of `*Flow`, we also introduce a simple adapter for executing Marple [40] traffic queries on GPV streams. The adapter, built on top of RaftLib [3], supports high level query primitives (map, filter, groupby, and zip) designed for operator-driven performance monitoring. Using `*Flow` along with the adapter allows operators to run many different queries concurrently, without having to compile them all to the PFE or pause the network to change queries. Analysis shows that the `*Flow` PFE pipeline requires only as many computational resources in the PFE as *one* compiled Marple query. Currently, the adapter scales to measure 15-50 Gb/s of traffic per core, bottlenecked only by overheads in our proof-of-concept implementation.

Contributions. This paper has four main contributions. First, the idea of using *grouped packet vectors* (GPVs) to lift the aggregation functions of traffic queries out of data plane hardware. Second, the design of a novel PFE cache data structure with dynamic memory allocation for efficient GPV generation. Third, the evaluation of a prototype of `*Flow` implemented on a readily available commodity P4 switch. Finally, four monitoring applications that demonstrate the practicality of `*Flow`.

[1] `https://github.com/jsonch/starflow`

	Efficient	Flexible	Concurrent	Dynamic
Netflow	✓	✗	✓	✓
Software	✗	✓	✓	✓
PFE Queries	✓	✓	✗	✗
***Flow**	✓	✓	✓	✓

Table 1: Practical requirements for PFE supported network queries.

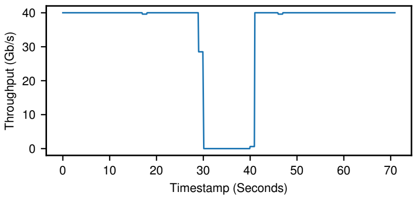


Figure 1: Network disruption from recompiling a PFE.

2 Background

In this section, we motivate the design goals of `*Flow` and describe prior telemetry systems.

2.1 Design Goals

*Flow is designed to meet four design goals that are important for a practical PFE accelerated telemetry system.

Efficient. We focus on efficient usage of processing servers in the telemetry and monitoring infrastructure of a network. Efficiency is important because telemetry and monitoring systems need to scale to high throughputs [50] and network coverage [32]. An inefficient telemetry system deployed at scale can significantly increase the total cost of a network, in terms of dollars and power consumption.

Flexible. A flexible telemetry system lets applications define the aggregation functions that compute traffic and data plane performance statistics. There are a wide range of statistics that are useful in different scenario and for different applications. Customizable aggregation functions allow a telemetry system to offer the broadest support.

Flexibility is also important for supporting future applications that may identify new useful metrics and systems that apply machine learning algorithms to analyze the network in many dimensions [43].

Concurrent. Concurrency is the capability to support many measurement queries at the same time. Concurrency is important because different applications require different statistics and, in a real network, there are likely to be many types of applications in use.

Consider a scenario where an operator is debugging an incast [15] and a network-wide security system is auditing for compromised hosts [36]. These applications would ideally run concurrently and need to measure different statistics. Debugging, for example, may benefit from measuring the number of simultaneously active TCP flows in a switch queue over small epochs, while a security application many require per-flow packet counters and timing statistics.

Dynamic. Support for dynamic queries is the capability to introduce or modify new queries at run time. It is important for monitoring applications, which may need to adapt as network conditions change, or themselves be launched at network run-time. Dynamic queries also enable interactive measurement [40] that can help network operators diagnose performance issues, e.g., *which queue is dropping packets between these hosts*?

2.2 Prior Telemetry Systems

Prior telemetry systems meet some, but not all, of the above design goals, as summarized in Table 1.

NetFlow Hardware. Many switches integrate hardware to generate NetFlow records [18] that summarize flows at the granularity of `IP` 5-tuple. NetFlow records are compact because they contain fully- computed aggregate statistics. ASICs [60, 20] in the switch data path do all the work of generating the records, so the overhead for monitoring and measurement applications is low. NetFlow is also dynamic. The ASICs are not embedded into the forwarding path, so a user can select different NetFlow features without pausing forwarding.

However, NetFlow sacrifices flexibility. Flow records have a fixed granularity and users choose statistics from a fixed list. Newer NetFlow ASICs [20] offer more statistics, but cannot support custom user-defined statistics or different granularities.

Software Processing. A more flexible approach is mirroring packets, or packet headers, to commodity servers that compute traffic statistics [19, 24, 22, 37]. Servers can also support concurrent and dynamic telemetry, as they are not in-line with data plane forwarding.

The drawback of software is efficiency. Two of the largest overheads for measurement in software are I/O [46], to get each packet or header to the measurement process, and hash table operations, to group packets by flow [50, 40, 33]. To demonstrate, we implemented a simple C++ application that reads packets from a PCAP, using lpcap, and computes the average packet length for each TCP flow. The application spent an average of 1535 cycles per packet on hash operations alone, using the relatively efficient C++ `std::unordered_map` [4]. In another application, which computed average packet length over pre-grouped vectors of packet lengths, the computation only took an average of 45 cycles per packet.

The benchmarks illustrate that mathematical opera-

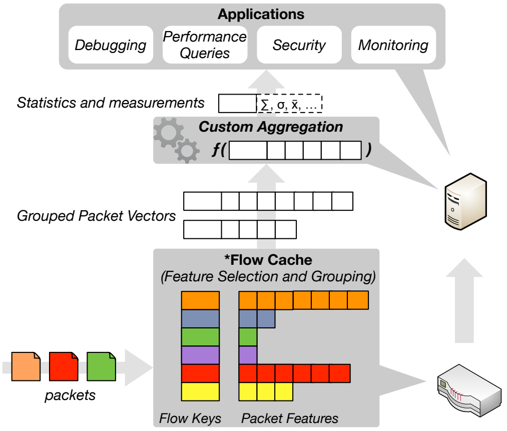


Figure 2: Overview of `*Flow`.

tions for computing aggregate statistics are *not* a significant bottleneck for measurement in software. Modern CPUs with vector instructions can perform upwards of 1 trillion floating point operations per second [39].

PFE Compiled Queries. Programmable forwarding engines (PFEs), the forwarding ASICs in next generation commodity switches [42, 13, 44, 41, 17, 53], are appealing for telemetry because they can perform stateful line-rate computation on packets. Several recent systems have shown that traffic measurement queries can compile to PFE configurations [50, 40]. These systems allow applications (or users) to define custom statistics computation functions and export records that include the aggregate statistics. Compiled queries provide efficiency and flexibility. However, they are not well suited for concurrent or dynamic measurement.

Concurrency is a challenge because of the processing models and computational resources available in a PFE. Each measurement query compiles to its own dedicated computational and memory resources in the PFE, to run in parallel at line rate. Computational resources are extremely limited, particularly those for stateful computation [49], making it challenging to fit more than a few queries concurrently.

Dynamic queries are a challenge because PFEs programs are statically compiled into configurations for the ALUs in the PFE. Adding a compiled query requires reloading the entire PFE program, which pauses all forwarding for multiple seconds, as Figure 1 shows. While it is possible to change forwarding rules at run-time to direct traffic through different pre-compiled functions, the actual computation can only be changed at compile time.

3 PFE Accelerated Telemetry with `*Flow`

`*Flow` is a PFE accelerated telemetry system that supports *efficient*, *flexible*, *concurrent*, and *dynamic* network measurement. It gains efficiency and flexibility by leveraging the PFE to select features from packet headers and group them by flow. However, unlike prior systems, `*Flow` lifts the complex and measurement-specific statistic computation, which are difficult to support in the PFE without limiting concurrent and dynamic measurement, up into software. Although part of the measurement is now in software, the feature selection and grouping done by the PFE reduces the I/O and hash table overheads significantly, allowing it to efficiently compute statistics and scale to terabit rate traffic using a small number of cores.

In this section, we overview the architecture of `*Flow`, depicted in Figure 2.

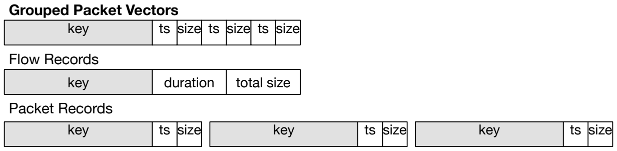


Figure 3: Comparison of grouped packet vectors, flow records, and packet records.

Grouped Packet Vectors. The key to decoupling feature selection and grouping from statistics computation is the *grouped packet vector* (GPV), a flexible and efficient format for telemetry data streamed from switches. A GPV stream is flexible because it contains per-packet features. Each application can measure different statistics from the same GPV stream and dynamically change measurement as needed, without impacting other applications or the PFE. GPVs are also efficient. Since the packet features are already extracted from packets and grouped, applications can compute statistics with minimal I/O or hash table overheads.

`*Flow` Telemetry Switches. Switches with programmable forwarding engines [42, 49] (PFEs) compile the `*Flow` cache to their PFEs to generates GPVs. The cache is implemented as a sequence of match+action tables that applies to packets at line rate and in parallel with other data plane logic. The tables extract features tuples from packets; insert them into per-flow GPVs; and stream the GPVs to monitoring servers, using multicast if there are more than 1.

GPV Processing. A thin `*Flow` agent running on a server receives GPVs from the switch and copies them to per-application queues. Each application defines its own statistics to compute over the packet tuples in GPVs and can dynamically change them as needed. Since the packet tuples are pre-grouped, the computation is extremely efficient because the bottleneck of mapping packets to flows is removed. Further, if fine granularity is needed, the applications can analyze the individual packet feature themselves, e.g., to identify the root cause of short lived congestion events, as Section 6.2 describes.

4 Grouped Packet Vectors (GPVs)

*Flow exports telemetry data from the switch in the grouped packet vector (GPV) format, illustrated in Figure 3, a new record format designed to support the decoupling of packet feature selection and grouping from aggregate statistics computation. A grouped packet vector contains an IP 5-tuple flow key and a variable length vector of feature tuples from sequential packets in the respective flow. As Figure 3 shows, a GPV is a hybrid between a packet record and a flow record. It inherits some of the best attributes of both formats and also has unique benefits that are critical for *Flow.

Similar to packet records, a stream of GPVs contains features from each individual packet. Unlike packet records, however, GPVs get the features to software in a format that is well suited for efficient statistics computation. An application can compute aggregate statistics directly on a GPV, without paying the overhead of receiving each packet, extracting features from it, or mapping it to a flow.

Similar to flow records, each GPV represents multiple packets and deduplicates the IP 5-tuple. They are around an order of magnitude smaller than packet header records and do not require software to perform expensive per-packet key value operations to map packet features to flows. Flow records are also compact and can be processed by software without grouping but, unlike flow records, GPVs do not lock the software into specific statistics. Instead, they allow the software to compute any statistics, efficiently, from the per-packet features. This works well in practice because many useful statistics derive from small, common subsets of packet features. For example, the statistics required by the 3 monitoring applications and 6 Marple queries we describe in Section 6 can all be computed from IP 5-tuples, packet lengths, arrival timestamps, queue depths, and TCP sequence numbers.

5 Generating GPVs

The core of *Flow is a cache that maps packets to GPVs and runs at line rate in a switch's programmable forwarding engine (PFE). A GPV cache would be simple to implement in software. However, the target platforms for *Flow are the hardware data planes of next-generation networks; PFE ASICs that process packets at guaranteed line rates exceeding 1 billion packets per second [49, 9, 16]. To meet chip space and timing requirements, PFEs significantly restrict stateful operations, which makes it challenging to implement cache eviction and dynamic memory allocation.

In this section, we describe the architecture and limitations of PFEs, cache eviction and memory allocation policies that can be implemented in a PFE, and our P4 implementation of the *Flow cache for the Tofino.

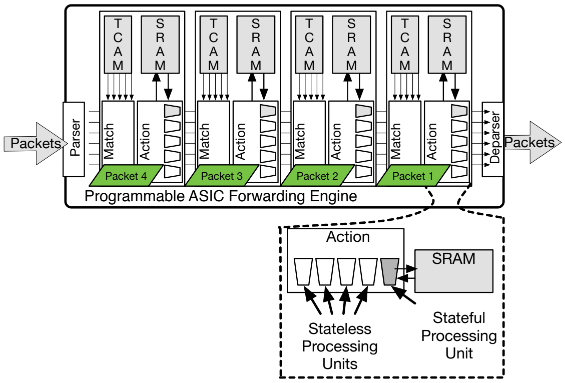


Figure 4: PFE architecture.

5.1 PFE Architecture

Figure 4 illustrates the general architecture of a PFE ASIC. It receives packets from multiple network interfaces, parses their headers, processes them with a pipeline of match tables and action processors, and finally deparses the packets and sends them to an output buffer. PFEs are designed specifically to implement match+action forwarding applications, e.g., P4 [8] programs, at guaranteed line rates that are orders of magnitude higher than other programmable platforms, such as CPUs, network processors [54], or FPGAs, assuming the same chip space and power budgets. They meet this goal with highly specialized architectures that exploit pipelining and instruction level parallelism [49, 9]. PFEs make it straightforward to implement custom terabit rate data planes, so long as they are limited to functionality that maps naturally to the match+action model, e.g., forwarding, access control, encapsulation, or address translation.

It can be challenging to take advantage of PFEs for more complex applications, especially those that require state persisting across packets, e.g., a cache. Persistent arrays, called "register arrays" in P4 programs, are stored in SRAM banks local to each action processor. They are limited in three important ways. First, a program can only access a register array from tables and actions implemented in the same stage. Second, each register array can only be accessed once per packet, using a *stateful ALU* that can implement simple programs for simultaneous reads and writes, conditional updates, and basic mathematical operations. Finally, the sequential dependencies between register arrays in the same stage are limited. In currently available PFEs [42], there can be no sequential dependencies; all of the registers in a stage must be accessed in parallel. Recent work, however, has demonstrated that future PFEs can ease this restriction to support pairwise dependencies, at the cost of slightly increased chip space [49] or lower line rates [16].

5.2 Design

To implement the *Flow cache as a pipeline of match+action tables that can compile to PFEs with the restrictions described above, we simplified the algorithms used for *cache eviction* and *memory allocation*. We do not claim that these are the best possible heuristics for eviction and allocation, only that they are intuitive and empirically effective starting points for a variable width flow cache that operates at multi-terabit line rates on currently available PFEs.

Cache Eviction. The *Flow cache uses a simple *evict on collision* policy. Whenever a packet from an untracked flow arrives and the cache needs to make room for a new entry, it simply evicts the entry of a currently tracked flow with the same hash value. This policy is surprisingly effective in practice, as prior work has shown [34, 50, 40], because it approximates a least recently used policy.

Memory Allocation. The *Flow cache allocates a narrow ring buffer for each flow, which stores GPVs. Whenever the ring buffer fills up, the cache flushes its contents to software. When an active flow fills its narrow buffer for the first time, the cache attempts to allocate a wider buffer for it, drawn from a pool with fewer entries than there are cache slots. If the allocation succeeds, the entry keeps the buffer until the flow is evicted; otherwise, the entry uses the narrow buffer until it is evicted.

This simple memory allocation policy is effective for *Flow because it leverages the long-tailed nature of packet inter-arrival time distributions [5]. In any given time interval, most of the packets arriving will be from a few highly active flows. A flow that fills up its narrow buffer in the short period of time before it is evicted is more likely to be one of the highly active flows. Allocating a wide buffer to such a flow will reduce the overall rate of messages to software, and thus its workload, by allowing the cache to accumulate more packet tuples in the ring buffer before needing to flush its contents to software.

This allocation policy also frees memory quickly once a flow's activity level drops, since frees happen automatically with evictions.

5.3 Implementation

Using the above heuristics for cache eviction and memory allocation, we implemented the *Flow cache as a pipeline of P4 match+action tables for the Tofino [42]. The implementation consists of approximately 2000 lines of P4 code that implements the tables, 900 lines of Python code that implements a minimal control program to install rules into the tables at runtime, and a large library that is autogenerated by the Tofino's compiler toolchain. The source code is available at our repository [2] and has been tested on both the Tofino's cycle-accurate simulator and a Wedge 100BF-32X.

Figure 5 depicts the control flow of the pipeline. It extracts a tuple of features from each packet, maps the tuple to a GPV using a hash of the packet's key, and then either appends the tuple to a dynamically sized ring buffer (if the packet's flow is currently tracked), or evicts the GPV of a prior flow, frees memory, and replaces it with a new entry (if the packet's flow is not currently tracked).

We implemented the evict on collision heuristic using a simultaneous read / write operations when updating the register arrays that store flow keys. The update action writes the current packet's key to the array, using its hash value as an index, and reads the data at that position into metadata in the packet. If there was a collision, which the subsequent stage can determine by comparing the packet's key with the loaded key, the remaining tables will evict and reset the GPV. Otherwise, the remaining tables will append the packet's features to the GPV.

We implemented the memory allocation using a stack. When a cache slot fills its narrow buffer for the first time, the PFE checks a stack of pointers to free extension blocks. If the stack is not empty, the PFE pops the top pointer from the stack. It stores the pointer in a register array that tracks which, if any, extension block each flow owns. For subsequent packets, the PFE loads the pointer from the array before updating its buffers. When the flow is evicted, the PFE removes the pointer from the array and pushes it back onto the free stack.

This design requires the cache to move pointers between the free stack and the allocated pointer array in both directions. We implemented it by placing the stack before the allocated pointer array, and resubmitting the packet to complete the free operation by pushing its pointer back onto the stack. The resubmission is necessary on the Tofino because sequentially dependent register arrays must be placed in different stages and there is no way to move "backwards" in the pipeline.

5.4 Configuration

Compile-time. The current implementation of the *Flow cache has three compile-time parameters: the number of cache slots; the number of entries in the dynamic memory pool; the width of the narrow and wide vectors; and the width of each packet feature tuple.

Feature tuple width depends on application requirements. For the other parameters, we implemented an OpenTuner [2] script that operates on a trace of packet arrival timestamps and a software model of the *Flow cache. The benchmarks in Section 7 show that performance under specific parameters is stable for long periods of time.

[2] https://github.com/jsonch/starflow

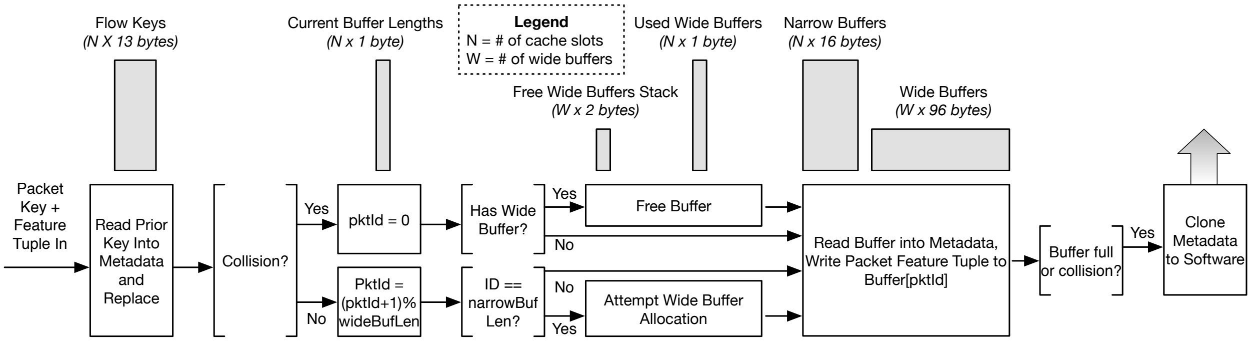


Figure 5: The `*Flow` cache as a match+action pipeline. White boxes represent sequences of actions, brackets represent conditions implemented as match rules, and gray boxed represent register arrays.

Run-time. The `*Flow` cache also allows operators to configure the following parameters at run-time by installing rules into P4 match+action tables. Immediately proceeding the `*Flow` cache, a filtering table lets operators install rules that determine which flows `*Flow` applies to, and which packet header and metadata fields go into packet feature tuples. After the `*Flow` cache, a table sets the destination of each exported GPV. The table can be configured to multicast GPVs to multiple servers and filter the GPV stream that each multicast group receives.

6 Processing GPVs

The `*Flow` cache streams GPVs to processing servers. There, measurement and monitoring applications (potentially running concurrently) can compute a wealth of traffic statistics from the GPVs and dynamically change their analysis without impacting the network.

In this section, we describe the `*Flow` agent that receives GPVs from the `*Flow` cache, three motivating `*Flow` monitoring applications, and the `*Flow` adapter to execute operator-driven network performance measurement queries on GPV streams.

6.1 The `*Flow` Agent

The `*Flow` agent, implemented as a RaftLib [3] application, reads GPV packets from queues filled by NIC drivers and pushes them to application queues. While applications can process GPVs directly, the `*Flow` agent implements three performance and housekeeping functions that are generally useful.

Load Balancing. The `*Flow` agent supports load balancing in two directions. First, a single `*Flow` agent can load balance a GPV stream across multiple queues to support applications that require multiple per-core instances to support the rate of the GPV stream. Second, multiple `*Flow` agents can push GPVs to the same queue, to support applications that operate at higher rates than a single `*Flow` agent can support.

GPV Reassembly. GPVs from a `*Flow` cache typically group packets from short intervals, e.g., under 1 second on average, due to the limited amount of memory available for caching in PFEs. To reduce the workload of applications, the `*Flow` agent can re-assemble the GPVs into a lower-rate stream of records that each represent a longer interval.

Cache Flushing. The `*Flow` agent can also flush the `*Flow` cache if timely updates are a priority. The `*Flow` agent tracks the last eviction time of each slot based on the GPVs it receives. It scans the table periodically and, for any slot that has not been evicted within a threshold period of time, sends a control packet back to the `*Flow` cache that forces an eviction.

6.2 *Flow Monitoring Applications

To demonstrate the practicality of `*Flow`, we implemented three monitoring applications that require concurrent measurement of traffic in multiple dimensions or packet-level visibility into flows. These requirements go beyond what prior PFE accelerated systems could support with compiled queries. With `*Flow`, however, they can operate efficiently, concurrently, and dynamically.

The GPV format for the monitoring applications was a 192 bit fixed width header followed by a variable length vector of 32 bit packet feature tuples. The fixed width header includes IP 5-tuple (104 bits), ingress port ID (8 bits), packet count (16 bits), and start timestamp (64 bits). The packet feature tuples include a 20 bit timestamp delta (e.g., arrival time - GPV start time), an 11 bit packet size, and a 1 bit flag indicating a high queue length during packet forwarding.

Host Timing Profiler. The host timing profiler generates vectors that each contain the arrival times of all packets from a specific host within a time interval. Such timing profiles are used for protocol optimizers [55], simulators [10], and experiments [52].

Prior to `*Flow`, an application would build these vectors by processing per- packet records in software, performing an expensive hash table operation to determine which host transmitted each packet.

With *Flow, however, the application only performs 1 hash operation per GPV, and simply copies timestamps from the feature tuples of the GPV to the end of the respective host timing vector. The reduction in hash table operations lets the application scale more efficiently.

Traffic Classifier. The traffic classifier uses machine learning models to predict which type of application generated a traffic flow. Many systems use flow classification, such as for QoS aware routing [23, 27], security [35, 51], or identifying applications using random port numbers or share ports. To maximize accuracy, these applications typically rely on feature vectors that contain dozens or even hundreds of different flow statistics [35]. The high cardinality is an obstacle to using PFEs for accelerating traffic classifiers, because it requires concurrent measurement in many dimensions.

*Flow is an ideal solution, since it allows an application to efficiently compute many features from the GPV stream generated by the *Flow cache. Our example classifier, based on prior work [43], measures the packet sizes of up to the first 8 packets, the means of packet sizes and inter-arrival times, and the standard deviations of packet size and inter-arrival times.

We implemented both training and classification applications, which use the same shared measurement and feature extraction code. The training application reads labeled "ground truth" GPVs from a binary file and builds a model using Dlib [30]; the classifier reads GPVs and predicts application classes using the model.

Micro-burst Diagnostics. This application detects micro-bursts [28, 48, 15], short lived congestion events in the network, and identifies the network hosts with packets in the congested queue at the point in time when the micro-burst occurred. This knowledge can help an operator or control application diagnose the root cause of periodic micro-bursts, e.g., TCP incasts [15], and also understand which hosts are affected by them.

Micro-bursts are difficult to debug because they occur at extremely small timescales, e.g., on the order of 10 microseconds [57]. At these timescales, visibility into host behavior at the granularity of individual packets is essential. Prior to *Flow, the only way for a monitoring system to have such visibility was to process a record from each packet in software [59, 25, 58, 56] and pay the overhead of frequent hash table operations.

With *Flow, however, a monitoring system can diagnose micro-bursts efficiently by processing a GPV stream, making it possible to monitor much more of the network without requiring additional servers.

The *Flow micro-burst debugger keeps a cache of GPVs from the most recent flows. When each GPV first arrives, it checks if the high queue length flag is set in any packet tuple. If so, the debugger uses the cached GPVs to build a globally ordered list of packet tuples, based on arrival timestamp. It scans the list backwards from the packet tuple with the high queue length flag to identify packet tuples that arrived immediately before it. Finally, the debugger determines the IP source addresses from the GPVs corresponding with the tuples and outputs the set of unique addresses.

6.3 Interactive Measurement Framework

An important motivation for network measurement, besides monitoring applications, is operator-driven performance measurement. Marple [40] is a recent system that lets PFEs accelerate this task. It presents a high level language for queries based around simple primitives (filter, map, group, and zip) and statistics computation functions. These queries, which can express a rich variety of measurement objectives, compile directly to the PFE, where they operate at high rates.

As discussed in Section 2, compiled queries make it challenging to support concurrent or dynamic measurement. Using *Flow, a measurement framework can gain the throughput benefits of PFE acceleration *without* sacrificing concurrency or dynamic queries, by implementing measurement queries in software, over a stream of GPVs, instead of in hardware, over a stream of packets.

To demonstrate, we extended the RaftLib [3] C++ stream processing framework with kernels that implement each of Marple's query primitives on a GPV stream. A user can define any Marple query by connecting the primitive kernels together in a connected graph defined in a short configuration file, similar to a Click [31] configuration file, but written in C++. The configuration compiles to a compact Linux application that operates on a stream of GPVs from the *Flow agent.

We re-wrote 6 example Marple queries from the original publication [40] as RaftLib configurations, listed in Table 4. The queries are functionally equivalent to the originals, but can all run concurrently and dynamically, without impacting each other or the network. These applications operate on GPVs with features used by the *Flow monitoring application, plus a 32 bit TCP sequence number in each packet feature tuple.

7 Evaluation

In this section, we evaluate our implementations of the *Flow cache, *Flow agent, and GPV processing applications. First, we analyze the PFE resource requirements and eviction rates of the *Flow cache to show that it is practical on real hardware. Next, we benchmark the *Flow agent and monitoring applications to quantify the scalability and flexibility benefits of GPVs. Finally, we

	Key Update	Memory Management	Pkt. Feature Update	**Total**
Computational				
Tables	3.8%	3.2%	17.9%	**25%**
sALUs	10.4%	6.3%	58.3%	**75%**
VLIWs	1.6%	1.1%	9.3%	**13%**
Stages	8.3%	12.5%	29.1%	**50%**
Memory				
SRAM	4.3%	1.0%	10.9%	**16.3%**
TCAM	1.1%	1.1%	10.3%	**12.5%**

Table 2: Resource requirements for `*Flow` on the Tofino, configured with 16384 cache slots, 16384 16-byte short buffers, and 4096 96-byte wide buffers.

compare the `*Flow` measurement query framework with Marple, to showcase `*Flow`'s support for concurrent and dynamic measurement.

All benchmarks were done with 8 unsampled traces from 10 Gbit/s core Internet routers taken in 2015 [11]. Each trace contained around 1.5 billion packets.

7.1 The `*Flow` Cache

We analyzed the resource requirements of the `*Flow` cache to understand whether it is practical to deploy and how much it can reduce the workload of software.

PFE Resource Usage. We analyzed the resource requirements of the `*Flow` cache configured with a tuple size of 32-bits, to support the `*Flow` monitoring applications, and a maximum GPV buffer length of 28, the maximum length possible while still fitting entirely into an ingress or egress pipeline of the Tofino. We used the tuning script, described in Section 5.4, to choose the remaining parameters using a 60 second trace from the 12/2015 dataset [12] and a limit of 1 MB of PFE memory.

Table 7.1 shows the computational and memory resource requirements for the `*Flow` cache on the Tofino, broken down by function. Utilization was low for most resources, besides stateful ALUs and stages. The cache used stateful ALUs heavily because it striped flow keys and packet feature vectors across the tofino's 32 bit register arrays, and each register array requires a separate sALU. It required 12 stages because many of the stateful operations were sequential: it had to access the key and packet count before attempting a memory allocation or free; and it had to perform the memory operation before updating the feature tuple buffer.

Despite the high sALU and stage utilization, it is still practical to deploy the `*Flow` cache alongside other common data plane functions. Forwarding, access control, multicast, rate limiting, encapsulation, and many other common functions do not require stateful operations, and so do not need sALUs. Instead, they need tables and SRAM, for exact match+action tables; TCAM, for longest prefix matching tables; and VLIWs, for modifying packet headers. These are precisely the resources that `*Flow` leaves free.

Further, the stage requirements of `*Flow` do not impact other applications. Tables for functions that are independent of `*Flow` can be placed in the same stages as the `*Flow` cache tables. The Tofino has high instruction parallelism and applies multiple tables in parallel, as long as there are enough computational and memory resources available to implement them.

PFE Resources Vs. Eviction Rate. Figure 6 shows the average packet and GPV rates for the Internet router traces, using the `*Flow` cache with the Tofino pipeline configuration described above. Shaded areas represent the range of values observed. An application operating on GPVs from the `*Flow` cache instead of packet headers needed to process under 18% as many records, on average, while still having access to the features of individual packets. The cache tracked GPVs for an average of 640MS and a maximum of 131 seconds. 14% of GPVs were cached for longer than 1 second and 1.3% were cached for longer than 5 seconds.

To analyze workload reduction with other configurations, we measured *eviction ratio*: the ratio of evicted GPVs to packets. Eviction ratio depends on the configuration of the cache: the amount of memory it has available; the maximum possible buffer length; whether it uses the dynamic memory allocator; and its eviction policy. We measured eviction ratio as these parameters varied using a software model of the `*Flow` cache. The software model allowed us to evaluate how `*Flow` performs on not only today's PFEs, but also on future architectures. We analyzed configurations that use up to 32 MB of memory, pipelines long enough to store buffers for 32 packet feature tuples, and hardware support for an 8 way LRU eviction policy. Larger memories, longer pipelines, and more advanced eviction policies are all proposed features that are practical to include in next generation PFEs [9, 16, 40].

Figure 7 plots eviction ratio as cache memory size varies, for 4 configurations of caches: with or without dynamic memory allocation; and with either a hash on collision eviction policy or an 8 way LRU. Division of memory between and buffer slots between the narrow and wide buffers was selected by the AutoTuner script. With dynamic memory allocation, the eviction ratio was between 0.25 and 0.071. This corresponds to an event rate reduction of between 4*X* and 14*X* for software, compared to processing packet headers directly.

On average, dynamic memory allocation reduced the amount of SRAM required to achieve a target eviction ratio by a factor of 2. It provided as much benefit as an 8

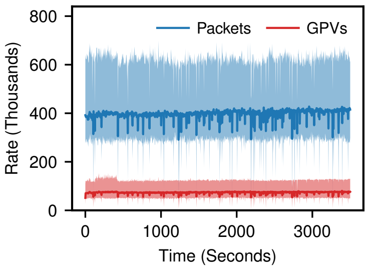


Figure 6: Min/avg./max of packet and GPV rates with *Flow for Tofino.

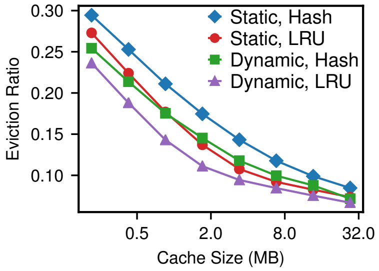


Figure 7: PFE memory vs eviction ratio.

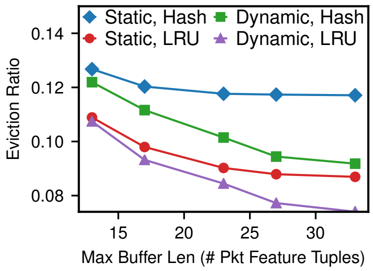


Figure 8: GPV buffer length vs eviction ratio.

# Cores	Agent	Profiler	Classifier	Debugger
1	0.60M	1.51M	1.18M	0.16M
2	1.12M	3.02M	2.27M	0.29M
4	1.85M	5.12M	4.62M	0.55M
8	3.07M	8.64M	7.98M	1.06M
16	3.95M	10.06M	11.43M	1.37M

Table 3: Average throughput, in GPVs per second, for *Flow agent and applications.

way LRU, but without requiring new hardware.

Figure 8 shows eviction rates as the maximum buffer length varied. Longer buffers required more pipeline stages, but significantly reduced eviction ratio when dynamic memory allocation was enabled.

7.2 *Flow Agent and Applications

We benchmarked the *Flow agent and monitoring applications, described in Section 6.2, to measure their throughput and quantify the flexibility of GPVs.

Experimental Setup. Our test server contained a Intel Xeon E5-2683 v4 CPU (16 cores) and 128 GB of RAM. We benchmarked maximum throughput by prepopulating buffers with GPVs generated by the *Flow cache. We configured the *Flow agent to read from these buffers and measured its throughput for reassembling the GPVs and writing them to a placeholder application queue. We then measured the throughput of each application individually, driven by a process that filled its input queue from a pre-populated buffer of reassembled GPVs. To benchmark multiple cores, we divided the GPVs across multiple buffers, one per core, that was each serviced by separate instances of the applications.

Throughput. Table 7.2 shows the average throughput of the *Flow agent and monitoring applications, in units of reassembled GPVs processed per second. For perspective, the average reassembled GPV rates for the 2015 10 Gbit/s Internet router traces, which are equal to their flow rates, are under 20 *thousand* per second [11].

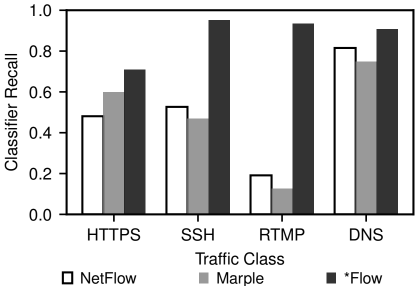


Figure 9: Recall of *Flow and baseline classifiers.

The high throughput makes it practical for a single server to scale to terabit rate monitoring. A server using 10 cores, for example, can scale to cover over 100 such 10 Gb/s links by dedicating 8 cores to the *Flow agent and 2 cores to the profiler or classifier.

Throughput was highest for the profiler and classifier. Both applications scaled to over 10 M reassembled GPVs per second, each of which contained an average of 33 packet feature tuples. This corresponds to a processing rate of over 300 M packet tuples per second, around 750X the average packet rate of an individual 10 Gb/s Internet router link.

Throughput for the *Flow agent and debugging application was lower, bottlenecked by associative operations. The bottleneck in the *Flow agent was the C++ `std::unordered_map` that it used to map each GPV to a reassembled GPV. The reassembly was expensive, but allowed the profiler and classifier to operate without similar bottlenecks, contributing to their high throughput.

In the debugger, the bottleneck was the C++ `std::map` it used to globally order packet tuples. In our benchmarks, we intentionally stressed the debugger by setting the high queue length flag in every packet feature tuple, forcing it to apply the global ordering function frequently. In practice, throughput would be much higher because high queue lengths only occur when there are problems in the network.

Configuration	# Stages	# Atoms	Max Width
`*Flow` cache	**11**	**33**	**5**
Marple Queries			
Concurrent Connections	4	10	3
EWMA Latencies	6	11	4
Flowlet Size Histogram	11	31	6
Packet Counts per Source	5	7	2
TCP Non-Monotonic	5	6	2
TCP Out of Sequence	7	14	4

Table 4: Banzai pipeline usage for the `*Flow` cache and compiled Marple queries.

Classifier Accuracy. To quantify the flexibility benefits of GPVs, we compared the `*Flow` traffic classifier to traffic classifiers that only use features that prior, less flexible, telemetry systems can measure. The *NetFlow* classifier uses metrics available from a traditional NetFlow switch: duration, byte count, and packet count. The *Marple classifier* also includes the average and maximum packet sizes as features, representing a query that compiles to use approximately the same amount of PFE resources as the `*Flow` cache.

Figure 9 shows the recall of the traffic classifiers on the 12/2015 Internet router trace. The `*Flow` classifier performed best because it had access to additional features from the GPVs. This demonstrates the inherent benefit of `*Flow`, and flexible GPV records, for monitoring applications that rely on machine learning and data mining. Also, as Table 7.2 shows, the classifier was performant enough to classify >1 million GPVs per second per core, making it well suited to live processing.

7.3 Comparison with Marple

Finally, to showcase `*Flow`'s support for concurrent and dynamic measurement, we compare the resource requirements for operator driven measurements using compiled Marple queries against the requirements using `*Flow` and the framework described in Section 6.3.

PFE Resources. For comparison, we implemented the `*Flow` cache for the same platform that Marple queries compile to: Banzai [49], a configurable machine model of PFE ASICs. In Banzai, the computational resources of a PFE are abstracted as *atoms*, similar to sALUs, that are spread across a configurable number of stages. The pipeline has a fixed width, which defines the number of atoms in each stage.

Table 4 summarizes the resource usage for the Banzai implementation. The requirements for `*Flow` were similar to those of a *single* statically compiled Marple query. Implementing all 6 queries, which represent only a small fraction of the possible queries, would require 79 atoms, over 2X more than the `*Flow` cache. A GPV stream contains the information necessary to support *all* the queries concurrently, and software can dynamically change them as needed without interrupting the network.

Server Resources. The throughput of the `*Flow` analytics framework was between 40 to 45K GPVs/s per core. This corresponded to a per-core monitoring capacity of 15 - 50 Gb/s, depending on trace. Analysis suggested that the bottleneck in our current prototype is message passing overheads in the underlying stream processing library that can be significantly optimized [38].

Even without optimization, the server resource requirements of the `*Flow` analytics framework are similar to Marple, which required around one 8 core server per 640 Gb/s switch [40] to support measurement of flows that were evicted from the PFE early.

8 Conclusion

Measurement is important for both network monitoring applications and operators alike, especially in large and high speed networks. Programmable forwarding engines (PFEs) can enable flexible telemetry systems that scale to the demands of such environments. Prior systems have focused on leveraging PFEs to scale efficiently with respect to throughput, but have not addressed the equally important requirement of scaling to support many concurrent applications with dynamic measurement needs. As a solution, we introduced `*Flow`, a PFE-accelerated telemetry system that supports dynamic measurement from many concurrent applications without sacrificing efficiency or flexibility. The core idea is to intelligently partition the query processing between a PFE and software. In support of this, we introduced GPVs, or grouped packet vectors, a flexible format for network telemetry data that is efficient for processing in software. We designed and implemented a `*Flow` cache that generates GPVs and operates at line rate on the Barefoot Tofino, a commodity 3.2 Tb/s P4 forwarding engine. To make the most of limited PFE memory, the `*Flow` cache features the first implementation of a dynamic memory allocator in a line rate P4 program. Evaluation showed that `*Flow` was practical in the switch hardware and enabled powerful GPV based applications that scaled efficiently to terabit rates with the capability for flexible, dynamic, and concurrent measurement.

Acknowledgements

We thank the anonymous reviewers for their input on this paper. This research was supported in part by the National Science Foundation under grants 1406192, 1406225, and 1406177 (SaTC) and 1652698 (CAREER); ONR under grant N00014-15-1-2006; and DARPA under contract HR001117C0047.

References

[1] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic flow scheduling for data center networks. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)* (2010), vol. 7, pp. 19–19.

[2] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation* (2014), ACM, pp. 303–316.

[3] Beard, J. C., Li, P., and Chamberlain, R. D. Raftlib: a c++ template library for high performance stream parallel processing. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores* (2015), ACM, pp. 96–105.

[4] Belton, J. Hash table shootout. `https://jimbelton.wordpress.com/2015/11/27/hash-table-shootout-updated/`.

[5] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (2010), ACM, pp. 267–280.

[6] Bhuyan, M. H., Bhattacharyya, D. K., and Kalita, J. K. Network anomaly detection: methods, systems and tools. *IEEE Communications Surveys & Tutorials 16*, 1 (2014), 303–336.

[7] Bjorner, N., Canini, M., and Sultana, N. Report on networking and programming languages 2017. *ACM SIGCOMM Computer Communication Review 47*, 5 (2017), 39–41.

[8] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (July 2014), 87–95.

[9] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 99–110.

[10] Botta, A., Dainotti, A., and Pescapé, A. Do you trust your software-based traffic generator? *IEEE Communications Magazine 48*, 9 (2010).

[11] CAIDA. Statistics for caida 2015 chicago direction b traces. `https://www.caida.org/data/passive/trace_stats/`, 2015.

[12] CAIDA. Trace statistics for caida passive oc48 and oc192 traces – 2015-12-17. `https://www.caida.org/data/passive/trace_stats/`, December 2015.

[13] Cavium. Cavium / xpliant cnx880xx product brief. `https://www.cavium.com/pdfFiles/CNX880XX_PB_Rev1.pdf?x=2`, 2015.

[14] Chen, A., Wu, Y., Haeberlen, A., Zhou, W., and Loo, B. T. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference* (2016), ACM, pp. 115–128.

[15] Chen, Y., Griffith, R., Liu, J., Katz, R. H., and Joseph, A. D. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking* (2009), ACM, pp. 73–82.

[16] Chole, S., Fingerhut, A., Ma, S., Sivaraman, A., Vargaftik, S., Berger, A., Mendelson, G., Alizadeh, M., Chuang, S.-T., Keslassy, I., et al. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 1–14.

[17] Cisco. The cisco flow processor: Cisco's next generation network processor solution overview. `http://www.cisco.com/c/en/us/products/collateral/routers/asr-1000-series-aggregation-services-routers/solution_overview_c22-448936.html`.

[18] Cisco. Introduction to cisco ios netflow. `https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html`, 2012.

[19] Cisco. Cisco netflow generation appliance 3340 data sheet. `http://www.cisco.com/c/en/us/products/collateral/cloud-systems-management/netflow-generation-3000-series-appliances/data_sheet_c78-720958.html`, July 2015.

[20] Cisco. Cisco nexus 9200 platform switches architecture. `https://www.cisco.com/c/dam/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-737204.pdf`, 2016.

[21] Claise, B. Cisco systems netflow services export version 9. `https://tools.ietf.org/html/rfc3954`, 2004.

[22] Deri, L., and SpA, N. nprobe: an open source netflow probe for gigabit networks. In *TERENA Networking Conference* (2003).

[23] Egilmez, H. E., Civanlar, S., and Tekalp, A. M. An optimization framework for qos-enabled adaptive video streaming over openflow networks. *IEEE Transactions on Multimedia 15*, 3 (2013), 710–715.

[24] Endace. Endaceflow 4000 series netflow generators. `https://www.endace.com/endace-netflow-datasheet.pdf`, 2016.

[25] Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., and McKeown, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), vol. 14, pp. 71–85.

[26] Heller, B., Seetharaman, S., Mahadevan, P., Yiakoumis, Y., Sharma, P., Banerjee, S., and McKeown, N. Elastictree: Saving energy in data center networks. In *NSDI* (2010), vol. 10, pp. 249–264.

[27] Hicks, M., Moore, J. T., Wetherall, D., and Nettles, S. Experiences with capsule-based active networking. In *DARPA Active NEtworks Conference and Exposition, 2002. Proceedings* (2002), IEEE, pp. 16–24.

[28] Jeyakumar, V., Alizadeh, M., Geng, Y., Kim, C., and Mazières, D. Millions of little minions: Using packets for low latency network programming and visibility. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 3–14.

[29] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 121–136.

[30] King, D. E. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research 10* (2009), 1755–1758.

[31] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The click modular router. *ACM Trans. Comput. Syst. 18*, 3 (Aug 2000), 263–297.

[32] Li, Y., Miao, R., Kim, C., and Yu, M. Flowradar: a better netflow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), pp. 311–324.

[33] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 1–13.

[34] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. Mica: A holistic approach to fast in-memory key-value storage. USENIX.

[35] Livadas, C., Walsh, R., Lapsley, D., and Strayer, W. T. Using machine learning technliques to identify botnet traffic. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on* (2006), IEEE, pp. 967–974.

[36] Lu, W., and Ghorbani, A. A. Network anomaly detection based on wavelet analysis. *EURASIP Journal on Advances in Signal Processing 2009* (2009), 4.

[37] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), USENIX Association, pp. 459–473.

[38] Michel, O., Sonchack, J., Keller, E., and Smith, J. M. Packet-level analytics in software without compromises. In *HotCloud* (2018).

[39] Microway. Detailed specifications of the skylake-sp intel xeon processor family. `https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-skylake-sp-intel-xeon-processor-scalable-family-cpus/`.

[40] Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., and Kim, C. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 85–98.

[41] Netronome. Agilio cx intelligent server adapters agilio cx intelligent server adapters. `https://www.netronome.com/products/agilio-cx/`, 2018.

[42] Networks, B. Barefoot tofino. `https://www.barefootnetworks.com/technology/#tofino`.

[43] Nguyen, T. T., and Armitage, G. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials 10*, 4 (2008), 56–76.

[44] Ozdag, R. Intel® ethernet switch fm6000 series-software defined networking, 2012.

[45] Phaal, P., Panchen, S., and McKee, N. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. Tech. rep., 2001.

[46] Rizzo, L., and Landi, M. Netmap: Memory Mapped Access to Network Devices. In *Proc. ACM SIGCOMM* (2011).

[47] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 123–137.

[48] Shan, D., Jiang, W., and Ren, F. Absorbing micro-burst traffic by enhancing dynamic threshold policy of data center switches. In *Computer Communications (INFOCOM), 2015 IEEE Conference on* (2015), IEEE, pp. 118–126.

[49] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 15–28.

[50] Sonchack, J., Aviv, A. J., Keller, E., and Smith, J. M. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 11:1–11:16.

[51] Sperotto, A., Schaffrath, G., Sadre, R., Morariu, C., Pras, A., and Stiller, B. An overview of ip flow-based intrusion detection. *IEEE communications surveys & tutorials 12*, 3 (2010), 343–356.

[52] Vanini, E., Pan, R., Alizadeh, M., Taheri, P., and Edsall, T. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *NSDI* (2017), pp. 407–420.

[53] Wheeler, B. A new era of network processing. *The Linley Group, Technical Report* (2013).

[54] Wheeler, B. A new era of network processing. *The Linley Group, Tech. Rep* (2013).

[55] Winstein, K., and Balakrishnan, H. Tcp ex machina: Computer-generated congestion control. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 123–134.

[56] Wu, Y., Chen, A., Haeberlen, A., Zhou, W., and Loo, B. T. Automated bug removal for software-defined networks. In *NSDI* (2017), pp. 719–733.

[57] Zhang, Q., Liu, V., Zeng, H., and Krishnamurthy, A. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference* (New York, NY, USA, 2017), IMC '17, ACM, pp. 78–85.

[58] Zhou, W., Sherr, M., Tao, T., Li, X., Loo, B. T., and Mao, Y. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 615–626.

[59] Zhu, Y., Kang, N., Cao, J., Greenberg, A., Lu, G., Mahajan, R., Maltz, D., Yuan, L., Zhang, M., Zhao, B. Y., et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 479–491.

[60] Zobel, D. Does my cisco device support netflow export? `https://kb.paessler.com/en/topic/5333-does-my-cisco-device-router-switch-support-netflow-export`, June 2010.

Applying Hardware Transactional Memory for Concurrency-Bug Failure Recovery in Production Runs

Yuxi Chen   Shu Wang   Shan Lu
University of Chicago

Karthikeyan Sankaralingam
University of Wisconsin – Madison

Abstract

Concurrency bugs widely exist and severely threaten system availability. Techniques that help recover from concurrency-bug failures during production runs are highly desired. This paper proposes BugTM, an approach that leverages Hardware Transactional Memory (HTM) on commodity machines for production-run concurrency-bug recovery. Requiring **no** knowledge about where are concurrency bugs, BugTM uses static analysis and code transformation to insert HTM instructions into multi-threaded programs. These BugTM-transformed programs will then be able to recover from a concurrency-bug failure by rolling back and re-executing the recent history of a failure thread. BugTM greatly improves the recovery capability of state-of-the-art techniques with low run-time overhead and no changes to OS or hardware, while guarantees not to introduce new bugs.

1 Introduction

1.1 Motivation

Concurrency bugs are caused by untimely accesses to shared variables. They are difficult to expose during in-house testing. They widely exist in production-run software [26] and have caused disastrous failures [23, 32, 40]. Production run failures severely hurt system availability: the restart after a failure could take long time and even lead to new problems if the failure leaves inconsistent system states. Furthermore, comparing with many other types of bugs, failures caused by concurrency bugs are particularly difficult to diagnose and fix correctly [50]. Techniques that handle production-run failures caused by concurrency bugs are highly desired.

Rollback-and-reexecution is a promising approach to recover failures caused by concurrency bugs. When a failure happens during a production run, the program rolls back and re-executes from an earlier checkpoint. Due to the unique non-determinism nature of concurrency bugs, the re-execution could get around the failure.

This approach is appealing for several reasons. It is generic, requiring no prior knowledge about bugs; it improves availability, masking the manifestation of concurrency bugs from end users; it avoids causing system inconsistency or wasting computation resources, which often come together with naive failure restarts; even if not successful, the recovery attempts only delays the failure by a negligible amount of time.

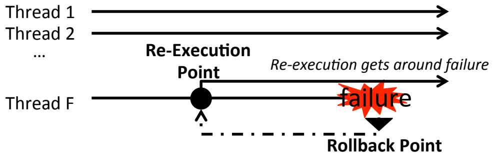


Figure 1: Single-threaded recovery for concurrency bugs

This approach also faces challenges in performance, recovery capability, and correctness (i.e., not introducing new bugs), as we elaborate below.

Traditional rollback recovery conducts full-blown multi-threaded re-execution and whole-memory checkpointing. It can help recover almost all concurrency-bug failures, but incurs too large overhead to be deployed in production runs [35, 39]. Even with support from operating systems changes, periodic full-blown checkpointing still often incurs more than 10% overhead [35].

A recently proposed recovery technique, ConAir, conducts single-threaded re-execution and register-only checkpointing [55]. As shown in Figure 1, when a failure happens at a thread, ConAir rolls back the register content of this thread through an automatically inserted `longjmp` and re-executes from the return of an automatically inserted `setjmp`, which took register checkpoints. This design offers great performance (<1% overhead), but also imposes severe limitations to failure-recovery capability. Particularly, with no memory checkpoints and re-executing only one thread, ConAir does not allow its re-execution regions to contain writes to shared variables (referred to as W_s) for correctness concerns, severely hurting its chance to recover many failures.

This limitation can be demonstrated by the real-world example in Figure 2. In this example, the `NULL` assignment from Thread-2 could execute between the write (A_1) and the read (A_2) on `s`→`table` from Thread-1, and cause failures. At the first glance, the failure could be recovered if we could rollback Thread-1 and re-execute both A_1 and A_2. However, such rollback and re-execution cannot be allowed by ConAir, as correctness can no longer be guaranteed if a write to a shared variable is re-executed (W_s in Figure 2): another thread t could have read the old value of s→table, saved it to a local pointer, the re-execution then gave s→table a new value, causing inconsistency between t and Thread-1 and deviation from the original program semantics.

```
//Thread-1                                //Thread-2
s->table = newTable(...); //A1, Ws
                                          s->table = NULL;
if(!s->table)              //A2
    //fatal-error message; software fails
```

Figure 2: A real-world concurrency bug from Mozilla

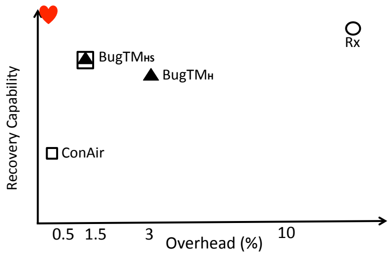


Figure 3: Design space of concurrency-bug failure recovery (Heart: non-existing optimal design; Rx [35] changes OS)

1.2 Contributions

Existing recovery techniques only touch two corners of the design space — good performance but limited recovery capability or good recovery capability but limited performance — as shown in Figure 3. It is desirable to have new recovery techniques that combine the performance and recovery capability strengths of the existing two corners of design, while maintaining correctness guarantees. BugTM provides such a new technique leveraging hardware transactional memory (HTM) support that already exists in commodity machines.

At the first glance, the opportunity seems obvious, as HTM provides a powerful mechanism for concurrency control and rollback-reexecution. Previous work [46] also showed that TM can be used to **manually** fix concurrency bugs **after** they are detected.

However, **automatically** inserting HTMs to help tackle **unknown** concurrency bugs during **production runs** faces many challenges not encountered by *manually* fixing *already detected* concurrency bugs *off-line*:

Performance challenges: High frequency of transaction uses would cause large overhead unacceptable for production runs. Unsuitable content of transactions, like trapping instructions[1], high levels of transaction nesting, and long loops, would also cause performance degradation due to repeated and unnecessary transaction aborts.

Correctness challenges: Unpaired transaction-start and transaction-commit could cause software to crash. Deterministic aborts, such as those caused by trapping instructions, could cause software to hang if not well handled. We need to guarantee these cases do not happen and ensure software semantics remains unmodified.

	ReExecution Point	RollBack Point	Checkpoint Memory ?	ReExecution contains W_s?
ConAir	setjmp	longjmp	✗	✗
BugTM$_H$	StartTx	AbortTx	✓	✓
BugTM$_{\text{HS}}$	setjmp or StartTx	longjmp or AbortTx	✓✗	✓

Table 1: Design comparisons (W_s: shared-variable writes)

Failure recovery challenges: In order for HTM to help recovery, we need to improve the chances that software executes in a transaction when a failure happens and we need to carefully design HTM-abort handlers to correctly process the corresponding transaction aborts.

BugTM addresses these challenges by its carefully designed and carefully inserted, based on static program analysis, HTM start, commit, and abort routines. Specifically, we have explored two BugTM designs: BugTM$_H$ and BugTM$_{\text{HS}}$, as highlighted in Table 1. They are both implemented as LLVM compiler passes that automatically instrument software in the following ways.

Hardware BugTM, short for BugTM$_H$, uses HTM techniques[2] *exclusively* to help failure recovery. When a failure is going to happen, a hardware transaction abort causes the failing thread to roll back. The re-execution naturally starts from the beginning of the enclosing transaction, carefully inserted by BugTM$_H$.

BugTM$_H$ provides better recovery capability than ConAir — benefiting from HTM, its re-execution region can contain shared variable writes. However, HTM costs more than setjmp/longjmp. Therefore, the performance of BugTM$_H$ is worse than ConAir, but much better than full-blown checkpointing, as shown in Figure 3.

Hybrid BugTM, short for BugTM$_{\text{HS}}$, uses HTM techniques and setjmp/longjmp *together* to help failure recovery. BugTM$_{\text{HS}}$ inserts both setjmp/longjmp and HTM APIs into software, with the latter inserted only when beneficial (i.e., when able to extend re-execution regions). When a failure is going to happen, the rollback is carried out through transaction abort if under an active transaction or longjmp otherwise.

BugTM$_{\text{HS}}$ provides performance almost as good as ConAir and recovery capability even better than BugTM$_H$ by carefully combining BugTM$_H$ and ConAir.

We thoroughly evaluated BugTM$_H$ and BugTM$_{\text{HS}}$ using 29 real-world concurrency bugs, including *all* the bugs used by a set of recent papers on concurrency bug detection and avoidance [17, 19, 41, 55, 56, 57]. Our evaluation shows that BugTM schemes can recover from

[1] Certain instructions such as system calls will deterministically cause HTM abort and are referred to as trapping instructions.

[2] This paper's implementation is based on Intel TSX. However, the principles apply to other vendors' HTM implementations.

many more concurrency-bug failures than state of the art, ConAir, while still provide good run-time performance — 3.08% and 1.39% overhead on average for BugTM$_H$ and BugTM$_{HS}$, respectively.

Overall, BugTM offers an easily deployable technique that can effectively tackle concurrency bugs in production runs, and presents a novel way of using HTM. Instead of using transactions to replace existing locks, BugTM automatically inserts transactions to harden the most failure-vulnerable part of a multi-threaded program, which already contains largely correct lock-based synchronization, with small run-time overhead.

2 Background

2.1 Transactional Memory (TM)

TM is a widely studied parallel programming construct [13, 15]. Developers can wrap a code region in a transaction (Tx), and the underlying TM system guarantees its atomicity, consistency, and isolation. Hardware transactional memory (HTM) provides much better performance than its software counterpart (STM), and has been implemented in IBM [12], Sun [8], and Intel commercial processors [1].

In this paper, we focus on Intel Transactional Synchronization Extensions (TSX). TSX provides a set of new instructions: `XBEGIN`, `XEND`, `XABORT`, and `XTEST`. We will denote them as `StartTx`, `CommitTx`, `AbortTx`, and `TestTx`, respectively for generality. Here, `CommitTx` may succeed or fail with the latter causing Tx abort. `AbortTx` explicitly aborts the current Tx, which leads to Tx re-execution unless special fallback code is provided. `TestTx` checks whether the current execution is under an active Tx.

There are multiple causes for Tx aborts in TSX. *Unknown abort* is mainly caused by trapping instructions, like exceptions and interrupts (abort code `0x00`). *Data conflict abort* is caused by conflicting accesses from another thread that accesses (writes) the write (read) set of the current Tx (abort code `0x06`). *Capacity abort* is due to out of cache capacity (abort code `0x08`). *Nested transaction abort* happens when there are more than 7 levels Tx nesting (abort code `0x20`). *Manual abort* is caused by `AbortTx` operation, with programmers specifying abort code.

2.2 ConAir

ConAir is a static code transformation tool built upon LLVM compiler infrastructure [22]. It is a state-of-the-art concurrency bug failure recovery technique as discussed in Section 1. We describe some techniques and terminologies that will be used in later sections below.

```
1 //Thread-1                            1 //Thread-2
2 if(thd->proc){    //A1                2
3     *buf++ = ' '; //Ws                3
4     strcat(buf,thd->proc);//A2        4 thd->proc = NULL;
5                       //failure site
6 }
```

Figure 4: A real-world concurrency bug from MySQL

Recovery capability limitations ConAir does not allow its re-execution regions to contain any writes to shared variables. Many of its re-execution points (i.e., `setjmps`) are put right after shared-variable writes, which prevent re-execution regions from growing longer and severely limit the recovery capability of ConAir.

ConAir fundamentally cannot recover **any** RAW[3] violations (e.g., the bug in Figure 2) and WAR violations, as Table 2 shows. The reason is that the (RA)W and W(AR) have to be re-executed for successful recoveries, but ConAir cannot re-execute shared-variable writes.

ConAir also cannot recover other types of concurrency bugs if a shared-variable write happens to exist between the failure location and the ideal re-execution point. For example, the RAR atomicity violation in Figure 4 cannot be recovered by ConAir due to the write to `*buf` on Line 3. If Line 3 did not exist, ConAir could have rolled back Thread-1 to re-execute Line 2 and gotten around the failure. With Line 3, ConAir can only repeatedly re-execute the `strcat` on Line 4, with no chance of recovery.

Failure instruction *f* ConAir automatically identifies where failures may happen so that rollback APIs can be inserted right there. This identification is based on previous observations that >90% of concurrency bugs lead to four types of failures [56]: assertion violations, segmentation faults, deadlocks, and wrong outputs. BugTM will reuse this technique to identify potential failure locations, denoted as *failure instructions f* in the remainder of the paper. Specifically, ConAir identifies the invocations of `__assert_fail` or other sanity-check macros as failure instructions for assertion failures. ConAir then automatically transforms software to turn segmentation faults and deadlocks into assertion failures: ConAir automatically inserts assertions to check whether a shared pointer variable *v* is `null` right before *v*'s dereference and check whether a pointer parameter of a string-library function is `null` right before the library call; ConAir automatically turns lock functions into time-out lock functions, with a long timeout indicating a likely deadlock failure, and inserts assertions accordingly. ConAir can help recover from wrong output failures as long as developers provide output specifications using assertions.

3 BugTM$_H$

3.1 High-Level Design

We discuss our high-level idea about where to put Txs, and compare with some strawman ideas based on perfor-

[3] (R/W)A(R/W) is short for (Read/Write)-after-(Read/Write).

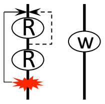
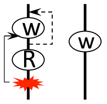
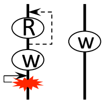
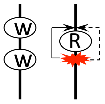
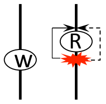
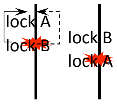
	Atomicity Violations				Order Violations	Deadlocks
	Read-after-Read (a) RAR	Read-after-Write (b) RAW	Write-after-Read (c) WAR	Write-after-Write (d) WAW	(e)	(f)
Types						
BugTM$_H$	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
ConAir	✓	–	–	✓	✓	✓

Table 2: Common types of concurrency bugs and how BugTM$_H$ and ConAir attempt to recover from them. (R/W: read/write to a shared variable; thick vertical line: the execution of one thread; dashed arrowed line: the re-execution region of BugTM$_H$; thin arrowed line: the re-execution region of ConAir; explosion symbol: a failure; -: cannot recover; ✓: sometimes can recover if the recovery does not require re-executing shared-variable writes; ✓✓: mostly can recover. The recovery procedure under BugTM$_{\text{HS}}$ is a mix of BugTM$_H$ and ConAir and hence is not shown in table.)

mance and failure-recovery capability.

Strawman approaches One approach is to chunk software to many segments and put every segment inside a hardware Tx [28]. This approach could avoid some atomicity violations, the most common type of concurrency bugs. However, it does not help recover from order violations, another major type of concurrency bugs. Furthermore, its excessive use of Txs will lead to unacceptable overhead for production-run deployment. Another approach is to replace all lock critical regions with Tx. However, this approach will not help eliminate many failures that are caused by missing lock.

Our approach In BugTM$_H$, we selectively put hardware Txs around places where failures may happen, like the invocation of an `__assert_fail`, the dereference of a shared pointer, etc. This design has the potential to achieve good performance because it inserts Txs only at selected locations. It also has the potential to achieve good recovery capability because in theory it can recover from all common types of concurrency bugs, as shown in Table 2 and explained below.

An atomicity violation (AV) happens when the atomicity of a code region $\mathbb{C}$ is unexpectedly violated, such as the bug shown in Figure 2. It contributes to more than 70% of non-deadlock concurrency bugs based on empirical studies [26], and can be further categorized into 4 sub-types depending on the nature of $\mathbb{C}$, as demonstrated in Table 2. Conflicting accesses would usually trigger a rollback recovery before the failure occurs, shown by the dashed arrow lines in Table 2(a)(b)(c), benefiting from the strong atomicity guarantee of Intel TSX — a Tx will abort even if the conflicting access comes from non-Tx code. For the bug shown in Figure 2 (an RAW atomicity violation), if we put the code region in Thread-1 inside a Tx, the interleaving NULL assignment from Thread-2 would trigger a data conflict abort in Thread-1 before the `if` statement has a chance to read the NULL. The re-execution of Thread-1 Tx will then re-assign the valid value to `s` → `table` for the `if` statement to read from, successfully avoiding the failure.

An order violation (OV) happens when an instruction *A* unexpectedly executes after, instead of before, instruction *B*, such as the bug in Figure 5. Different from AVs, conflicting memory accesses related to OVs may not all happen inside a small window. In fact, *A* may not have executed when a failure occurs in the thread of *B*. Consequently, the Tx abort probably will be triggered by a software failure, instead of a conflicting access, depicted by the dashed arrow in Table 2(e). Fortunately, the rollback reexecution will still give the software a chance to correct the unexpected ordering and recover from the failure. Take the bug shown in Figure 5 as an example. If we put a hardware Tx in Thread-1, when order violation leads to the assertion failure, the Tx will abort, rollback, and re-execute. Eventually, the pointer `ptr` will be initialized and the Tx will commit.

```
//Thread-1                      //Thread-2
                                //ptr is NULL until
assert (ptr); //B               //initialized at A
//should execute after A        ptr = malloc (K); //A
```

Figure 5: A real-world OV bug (simplified from `Transmission`)

Deadlock bugs occur when different threads each holds resources and circularly waits for each other. As shown in Table 2(f), it can be recovered by Tx rollback and re-execution too, as long as deadlocks are detected.

Of course, BugTM$_H$ cannot recover from *all* failures, because some error-propagation chains cannot fit into a HTM Tx, which we will discuss more in Section 7.

Next, we will discuss in details how BugTM$_H$ surrounds failure sites with hardware Txs— how to automatically insert `StartTx`, `CommitTx`, `AbortTx`, and fallback-/retry code into software, while targeting three goals: (1) good recovery capability; (2) good run-time performance; (3) not changing original program semantics.

3.2 Design about `AbortTx`

BugTM$_H$ uses the same technique as ConAir to identify where failures would happen as discussed in Sec-

```
        if(_xtest()){
                //manually abort with abort code 0xFF
                _xabort(0xFF);
        }
```

Figure 6: BugTM$_H$`AbortTx` wrapper function (`my_xabort`)

tion 2.2. BugTM$_H$ puts an `AbortTx` wrapper function `my_xabort` right before every failure instruction f, so that a Tx abort and re-execution is triggered right before a failure manifests. `my_xabort` uses a unique abort code `0xFF` for its `AbortTx` operation (as shown in Figure 6), so that BugTM$_H$ can differentiate different causes of Tx aborts and handle them differently.

3.3 Design about `StartTx` and `CommitTx`

Challenges We elaborate on two key challenges in placing `StartTx` and `CommitTx`, and explain why we can**not** simply insert well-structured atomic blocks (e.g., `__transaction_atomic` supported by GCC) into programs.

First, poor placements could cause frequent Tx aborts. Trapping instructions (e.g., system calls) and heavy TM nesting (>7 level) deterministically cause aborts, while long Txs abort more likely than short ones due to timer-interrupts and memory-footprint threshold. These aborts hurt not only performance, but also recovery — deterministic aborts of a Tx will eventually force us to execute the Tx region[4] in non-transaction mode, leaving no hope for failure recovery.

Second, poor placements could cause unpaired execution of `StartTx` and `CommitTx`, hurting both correctness and performance. When `CommitTx` executes without `StartTx`, the program will crash; when `StartTx` executes without a pairing `CommitTx`, its Tx will repeatedly abort.

Taking Figure 7 as an example, we want to put A_1 and A_2, both accessing global variable `G`, into a Tx together with `__assert_fail` on Line 6 for failure recovery. However, if we naively put `StartTx` on Line 2 and `CommitTx` on Line 12, forming a well structured atomic block, correct runs will incur repeated Tx aborts and huge slowdowns due to I/Os on Line 10. Simply moving `CommitTx` to right after Line 4 and keeping `StartTx` on Line 2 still will not work — when `else` is taken, the earlier `StartTx` has no pairing `CommitTx` and the Tx still aborts due to I/Os.

We address the first challenge by carefully placing `StartTx` and `CommitTx`. We address the second challenge mainly through our `StartTx`, `CommitTx` wrapper-functions.

```
 1 void func(...){                              1 void func(...){
 2                                              2 + my_xbegin();
 3   G = g;  //A1                               3   G = g;
 4   if(!G){         //A2                       4   if(!G){
 5                                              5 +   my_xabort();
 6     __assert_fail;//f: failure instr.        6     __assert_fail;
 7   }                                          7   }
 8   else{                                      8   else{
 9                                              9 +   my_xend();
10     IO(...);   //computation & I/O          10     IO(...);
11   }                                         11   }
12                                             12 + my_xend();
13 }                                           13 }
```

Figure 7: A toy example adapted from Figure 2 (left-side) and its BugTM$_H$ transformation (right-side)

```
    if(_xtest() == 0){//no active Tx
        Retrytimes = 0;
        prev_status = -1;
retry:  if((status = _xbegin()) == _XBEGIN_STARTED){
          //Tx starts
        }else{
          //abort fallback handler, no active Tx at this point
          Retrytimes++;
          if(status==0x00||status==0x08){
          //unknown or capacity abort
            if(!(prev_status==0x00 && status==0x00) &&
                !(prev_status==0x08 && status==0x08))
             { prev_status=status; goto retry;}
           }else if(status==0x06 || status==0xFF){
             if(Retrytimes < RetryThreshold)
               {prev_status=status; goto retry;}
           }
           //continue execution in non-Tx mode
        }
    }
```

Figure 8: BugTM$_H$`StartTx` wrapper function (`my_xbegin`)

Where to `StartTx` and `CommitTx` The design principle is to minimize the chance of aborts that are unrelated to concurrency bugs, tackling the first challenge above. BugTM$_H$ achieves this by making sure that its Txs do not contain function calls, which avoids system calls and many trapping instructions, or loops, which avoids large memory footprints. The constraint of not containing function calls will be relaxed in Section 3.5.

Specifically, for every failure instruction f inside a function F, BugTM$_H$ puts a `StartTx` wrapper function right after the first function call instruction or loop-exit instruction or the entrance of F, whichever encountered first along every path tracing backward from f to the entrance of F. BugTM$_H$ puts `CommitTx` wrapper functions right before the exit of F, every function call in F, and every loop header instruction in F, unless the corresponding loop contains a failure instruction, in which case we want to extend re-execution regions for possible failures inside the loop.

Analysis for different failure instructions may decide to put multiple `StartTx` (`CommitTx`) at the same program location. In these cases, we will only keep one copy.

For the toy example in Figure 7, the intra-procedural BugTM$_H$ identifies Line 2 to put a `StartTx`, and identifies Line 9 and 12 to put `CommitTx`, as shown in the figure.

[4] We will refer to the code region between our `my_xbegin` and `my_xend` as a Tx region, which may be executed in transactional mode.

```
if(_xtest())
    _xend(); //terminate an active transaction
```

Figure 9: BugTM$_H$CommitTx wrapper function (my_xend)

How to StartTx and CommitTx The above algorithm does not guarantee one-to-one pairing of the execution of StartTx and CommitTx, the second challenge discussed above. BugTM$_H$ addresses this through TestTx checkings conducted in my_xbegin and my_xend, BugTM$_H$ wrapper functions for StartTx and CommitTx. That is, StartTx will execute only when there is no active Txs, as shown in Figure 8; CommitTx will execute only when there exists an active Tx, as shown in Figure 9.

Overall, our design so far satisfies performance, correctness, and failure-recovery goals by guaranteeing a few properties. For performance, BugTM$_H$ guarantees that its Txs do not contain system/library calls or loops or nested Txs, and always terminate by the end of the function where the Tx starts. For correctness, BugTM$_H$ guarantees not to introduce crashes caused by unpairing CommitTx. For recovery capability, BugTM$_H$ makes the best effort in letting failures occur under active Txs.

3.4 Design for fallback and retry

Challenges It is not trivial to automatically and correctly generate fallback/retry code for all Txs inserted by BugTM$_H$. Since many Tx aborts may be unrelated to concurrency bugs, inappropriate abort handling could lead to performance degradation, hangs, and lost failure-recovery opportunities.

Solutions BugTM$_H$ will check the abort code and react to different types of aborts differently. Specifically, BugTM$_H$ implements the following fallback/retry strategy through its my_xbegin wrapper (Figure 8).

Aborts caused by AbortTx inserted by BugTM$_H$ indicates software failures. We should re-execute the Tx under HTM, hoping that the failure will disappear in retry (Line 14–17). To avoid endless retry, BugTM$_H$ keeps a retry-counter Retrytimes (Figure 8). This counter is configurable in BugTM$_H$, with the default being 1000000.

Data conflict aborts (Line 14–17) are caused by conflicting accesses from another thread. They are handled in the same way as above, because they could be part of the manifestation of concurrency bugs.

Unknown aborts and capacity aborts (Line 9–13) have nothing to do with concurrency bugs or software failures. In fact, the same abort code may appear repeatedly during retries, causing performance degradation without increasing the chance of failure recovery. Therefore, the fallback code will re-execute the Tx region in non-transaction mode once these two types of aborts are observed in two consecutive aborts. Nested Tx aborts would not be encountered by BugTM$_H$, because BugTM$_H$ Txs are non-nested.

The above wrapper function not only implements fallback/retry strategy, but also allows easy integration into the target software, as demonstrated in Figure 7.

3.5 Inter-procedural Designs and Others

The above algorithm allows no function calls or returns in Txs, keeping the whole recovery attempt within one function F. This is too conservative as many functions contain no trapping instructions and could help recovery.

To extend the re-execution region into callees of F, we put my_xend before every system/library call instead of every function call. To extend the re-execution region into the callers of F, we slightly change the policy of putting my_xbegin. When the basic algorithm puts my_xbegin at the entrance of F, the inter-procedural extension will find all possible callers of F, treat the callsite of F in its caller as a failure instruction, and apply my_xbegin insertion and my_xend insertion in the caller.

We then adjust our strategy about when to finish a BugTM$_H$ Tx. The basic BugTM$_H$ may end a Tx too early: by placing my_xend before every function exit, the re-execution will end in a callee function of F before returning to F and reaching the potential failure site in F. Our adjustment changes the my_xend wrapper inserted at function exits, making it take effect only when the function is the one which starts the active Tx.

Finally, as an optimization, we eliminate Txs that contain no shared-variable reads the failure instruction f has control or data dependency on. In these cases, the execution and outcome of f is deterministic during re-execution, and hence the failure cannot be recovered.

4 BugTM$_{HS}$

Rollback and re-execution techniques based on HTM (Section 3) and setjmp/longjmp [55] each has its own strengths and weaknesses. The former allows re-execution regions to contain shared variable writes, which is a crucial improvement over the latter in terms of failure recovery capability. However, it also has higher overhead than the latter. Furthermore, some operations not allowed inside an HTM Tx (e.g. malloc, memcpy, pthread_cond_wait), could potentially be correctly re-executed through software techniques [37, 45].

To combine the strengths of the above two approaches, we design BugTM$_{HS}$. The high level idea is that we apply ConAir to insert setjmp and longjmp recovery code into a program first[5]; and then, only at places where the growth of re-execution regions are stopped by shared-variable writes, we apply BugTM$_H$ to extend re-execution regions through HTM-based recovery.

[5] Intel TSX allows setjmp/longjmp to execute inside Txs.

Next, we will discuss in details how we carry out this high level idea to achieve the **union** of BugTM$_H$ and ConAir's recovery capability, while greatly enhancing the performance of BugTM$_H$.

Where to setjmp and StartTx ConAir and BugTM$_H$ insert setjmp and StartTx using similar algorithms, easing the design of BugTM$_{\text{HS}}$. That is, for every failure instruction f inside a function F, ConAir (BugTM$_H$) traverses backward through every path p that connects f with the entrance of F on CFG, and puts a setjmp wrapper function (StartTx wrapper function) right after the first appearance of a *killing instruction*. We will refer to this location as loc$_{\text{setjmp}}$ and loc$_{\text{StartTx}}$, respectively. For ConAir, the killing instructions include the entrance of F, writes to any global or heap variables, and a selected set of system/library calls; for BugTM$_H$, the killing instructions include the entrance of F, the loop-exit instruction, and all system/library calls [6].

BugTM$_{\text{HS}}$ slightly modifies the above algorithm. Along every path p, BugTM$_{\text{HS}}$ inserts the setjmp wrapper function at every loc$_{\text{setjmp}}$, where ConAir would insert it. In addition, BugTM$_{\text{HS}}$ inserts the StartTx wrapper function at loc$_{\text{StartTx}}$, when loc$_{\text{StartTx}}$ is farther away from f than loc$_{\text{setjmp}}$ (i.e., offering longer re-execution). Note that BugTM$_{\text{HS}}$ inserts setjmp at every location loc$_{\text{setjmp}}$ where ConAir would have inserted setjmp because every loc$_{\text{setjmp}}$ might be executed without an active hardware transaction due to unexpected HTM aborts and others. When loc$_{\text{setjmp}}$ is same as loc$_{\text{StartTx}}$, BugTM$_{\text{HS}}$ would only insert setjmp without inserting StartTx wrapper function.

Where to CommitTx BugTM$_{\text{HS}}$ inserts CommitTx wrapper functions exactly where BugTM$_H$ inserts them. Note that, BugTM$_{\text{HS}}$ inserts fewer StartTx than BugTM$_H$, and hence starts fewer Txs at run time. Fortunately, this does not affect the correctness of how BugTM$_{\text{HS}}$ inserts CommitTx, because the wrapper function makes sure that CommitTx executes only under an active Tx.

How to retry ConAir and BugTM$_H$ insert longjmp and AbortTx wrapper functions, which are responsible for triggering rollback-based failure recovery, using the same algorithm — right before a failure is going to happen as described in Section 2.2 and Section 3.2.

BugTM$_{\text{HS}}$ inserts its rollback function (Figure 10) at the same locations. We design BugTM$_{\text{HS}}$ rollback wrapper to first invoke HTM-rollback (i.e., AbortTx) if it is under an active transaction, which will allow a longer re-execution region and hence a higher recovery probability. The BugTM$_{\text{HS}}$ rollback wrapper invokes longjmp rollback if it is not under an active transaction. To make sure that the program would not keep attempting hopeless recoveries, BugTM$_{\text{HS}}$ continues to use the HTM-abort statistics in the StartTx wrapper function shown in Figure 8 and continues to keep the longjmp retry count threshold shown in Figure 10.

```
if(_xtest())
  _xabort(0xFF); //terminate an active transaction
else //use longjmp for recovery
    if(longjmp_retry ++ < 1000000) // avoid endless retry
        longjmp(buf1,-1);
```

Figure 10: BugTM$_{\text{HS}}$ rollback wrapper function

For examples shown in Figure 2, 4, and 7, BugTM$_{\text{HS}}$ would insert both setjmp and StartTx into the buggy code regions, because StartTx would provide longer re-execution regions in all three cases. However, if the *buf++ = ' '; statement does not exist in Figure 4, BugTM$_{\text{HS}}$ would not insert StartTx there. Consequently, if failures happen, longjmp will be used for recovery.

Overall, we expect BugTM$_{\text{HS}}$ to improve the performance of BugTM$_H$ and improve the recovery capability of both BugTM$_H$ and ConAir. This will be confirmed through experiments in Section 7.

5 Failure Diagnosis

Previous recovery techniques like ConAir and naive system restart leave failure diagnosis completely to developers, which is often very time consuming. To address this limitation, we design BugTM$_{\text{HS}}$ to support failure diagnosis through the root-cause inference routine shown in Figure 11 and extra logging during recovery.

The root-cause inference shown in Figure 11 is mostly straightforward. The rationale of diagnosis based on the number of re-executions (Line 5 and 7) is the following. If the recovery success relies on a code region $\mathbb{C}$ in the failure thread to re-execute atomically, probably one re-execution attempt is sufficient, because another unserializable interleaving during re-execution is very rare. This case applies to RAR violation, as shown in Table 2. If the recovery success relies on something to happen in another thread, multiple re-executions are probably needed. This applies to WAW violations and order violations, as shown in Table 2.

Note that, BugTM$_{\text{HS}}$ and BugTM$_H$ could detect and recover the software from concurrency bugs before explicit failures getting triggered. As shown in Table 2, for several types of atomicity violation bugs, the retry would be triggered by HTM data-conflict aborts, instead of explicit failures. In these cases (Line 9), BugTM$_{\text{HS}}$ cannot affirmatively conclude that concurrency bugs have happened. It can only provide hints that certain types of atomicity violations may be the reason for HTM aborts. Along this line, future work could extend BugTM to contain more concurrency-bug detection capability, in addi-

[6] BugTM$_{\text{HS}}$ also combines the inter-procedural recovery of ConAir and BugTM$_H$ in a similar way. We skip details for space constraints.

```
Input: information from a successful recovery
if (timeout failures)
    output: deadlock
else if (other explicit failures)
    if (first re-execution succeeds)
      output: RAR atomicity violation
    else
      output: Order Violation or WAW atomicity violation
else if (implicit failures) //HTM data conflict aborts
    output: possible RAR, WAR, or RAW atomicity violations
```

Figure 11: Recovery-guided root-cause diagnosis

tion to its failure recovery capability.

BugTM$_{HS}$ also logs memory access type (read/write), addresses, values, and synchronization operations during re-execution, which helps diagnosis with no run-time overhead and only slight recovery delay.

Of course, some real-world concurrency bugs are complicated. However, complicated bugs can often be decomposed into simpler ones. Furthermore, some principles still hold. For example, if the re-execution succeeds with just one attempt, it is highly likely that an atomicity violation happened to the re-execution region.

6 Methodology

Implementation BugTM is implemented using LLVM infrastructure (v3.6.1). We obtained the source code of ConAir, also built upon LLVM. All the experiments are conducted on 4-core Intel Core i7-5775C (Broadwell) machines with 6MB cache, 8GB memory running Linux version 2.6.32, and O3 optimization level.

Benchmark suite We have evaluated BugTM on 29 bugs, including *all* the real-world bug benchmarks in a set of previous papers on concurrency-bug detection, fixing, and avoidance [17, 19, 41, 55, 56, 57]. They cover all common types of concurrency-bug root causes and failure symptoms. They are from server applications (e.g., MySQL database server, Apache HTTPD web server), client applications (e.g., Transmission BitTorrent client), network applications (e.g., HawkNL network library, HTTrack web crawler, Click router), and many desktop applications (e.g., PBZIP2 file compressor, Mozilla JavaScript Engine and XPCOM). The sizes of these applications range 50K — 1 million lines of code. Finally, our benchmark suite contains 3 extracted benchmarks: Moz52111, Moz209188, and Bank.

The goal of BugTM is to *recover* from production-run failures, **not** to *detect* bugs. Therefore, our evaluation uses previously known concurrency bugs that we know how to trigger failures. In all our experiments, the evaluated recovery tools do **not** rely on any knowledge about specific bugs in their failure recovery attempts.

Setups and metrics We will measure the recovery capability and overhead of BugTM$_H$ and BugTM$_{HS}$. We will also evaluate and compare with ConAir [55], the state of the art concurrency-bug recovery technique.

	RootCause	ConAir	BugTM$_H$	BugTM$_{HS}$
MySQL2011	AV$_{RAR}$	–	✓	✓
MySQL38883	AV$_{RAR}$	–	✓	✓
Apache21287	AV$_{RAW}$	–	✓	✓
Moz-JS18025	AV$_{RAW}$	–	✓	✓
Moz-JS142651	AV$_{RAW}$	–	✓	✓
Bank	AV$_{WAR}$	–	✓	✓
Transmission	OV	✓	–	✓
Total		1	6	7

Table 3: Recovery capability comparison (Moz-JS: Mozilla JavaScript Engine.)

To measure recovery capability, we follow the methodology of previous work [18, 55], and insert `sleeps` into software, so that the corresponding bugs will manifest frequently. We then run each bug-triggering workload with each tool applied for 1000 times.

To measure the run-time overhead. We run the original software **without** any `sleeps` with each tool applied. We report the average overhead measured during 100 failure-**free** runs, reflecting the performance during **regular** execution. We also evaluate alternative designs of BugTM, such as not conducting inter-procedural recovery, not excluding system calls from Txs, not excluding loops, etc. Due to space constraints, we only show this set of evaluation results on Mozilla and MySQL benchmarks, two widely used client and server applications.

7 Experimental Results

Overall, BugTM$_H$ and BugTM$_{HS}$ both have better recovery capability than ConAir, and both provide good performance. BugTM$_{HS}$ provides the best combination of recovery capability and performance among the three.

7.1 Failure recovery capability

Among all the 29 benchmarks, 9 cannot be recovered by any of the evaluated techniques, no matter ConAir or BugTM, and the remaining 20 can be recovered by at least one of the techniques (BugTM$_{HS}$ can recover all of these 20). Table 3 shows the result of 7 benchmarks where different tools show different recovery capability.

ConAir fails to recover from 6 out of 7 failures in Table 3, mainly because it does not allow shared-variable writes in re-execution regions. As a result, it cannot recover from any RAW or WAR atomicity bugs, and some RAR bugs, including the one in Figure 4.

BugTM$_H$ can successfully recover from all the 6 failures that ConAir cannot in Table 3. BugTM$_H$ cannot recover from the Transmission bug, because recovering this bug requires re-executing `malloc`, a trapping operation for Intel TSX but handled by ConAir. In fact, `malloc` is allowed in some more sophisticated TM designs [37, 45].

BugTM$_{HS}$ combines the strengths of BugTM$_H$ and ConAir, and hence can successfully recover from all 7

	Run-time Overhead			`#setjmp`	`#StartTx`		`#StartTx` per 10μs		Abort%	
	ConAir	BugTM$_H$	BugTM$_{HS}$	BugTM$_{HS}$	BugTM$_H$	BugTM$_{HS}$	BugTM$_H$	BugTM$_{HS}$	BugTM$_H$	BugTM$_{HS}$
MySQL2011	0.05%	0.13%	0.08%	643425	2746031	778024	2.3	0.7	0.01	0.01
MySQL3596	0.40%	3.10%	1.12%	144212	110476	39913	3.9	1.4	0.12	0.20
MySQL38883	0.40%	3.08%	1.11%	144119	110471	39904	3.9	1.4	0.11	0.19
Apache21287	0.55%	3.77%	3.00%	40023	72093	45520	22.8	14.5	0.08	0.11
Moz-JS18025	0.57%	9.03%	2.62%	3992	6850	1159	16.3	2.8	0.29	0.04
Moz-JS142651	0.76%	11.9%	5.30%	2145	9666	4007	30.4	12.6	0.33	0.17
Bank	0.15%	2.18%	2.95%	6	5	5	0.1	0.1	0.0	0.0
Moz-ex52111	0.47%	0.53%	0.41%	4	3	0	0.0	0.0	0.0	0.0
Moz-ex209188	0.12%	0.58%	0.77%	2	1	1	0.0	0.0	0.0	0.0
MySQL791	0.35%	1.98%	0.24%	48998	4948	602	2.5	0.4	0.35	0.01
MySQL16582	0.15%	3.03%	0.99%	269543	153532	31222	3.8	0.8	0.03	0.06
Click	0.57%	8.11%	3.60%	4681	5142	2123	18.7	8.1	0.96	0.12
FFT	0.05%	0.03%	0.14%	23	25	19	0.0	0.0	0.0	0.0
HTTrack	0.15%	0.64%	0.04%	9212	15649	1572	0.1	0.0	0.83	0.11
Moz-xpcom	0.38%	0.45%	0.03%	324	1933	154	0.0	0.0	0.31	0.51
Transmission	0.11%	0.22%	0.07%	1093	2123	919	0.1	0.0	0.56	0.40
zsnes	0.05%	0.03%	0.44%	10462	11737	372	0.5	0.0	0.13	0.23
HawkNL	0.09%	0.00%	0.15%	10	19	16	0.0	0.0	0.0	0.07
Moz-JS79054	0.84%	11.7%	4.20%	338	1325	360	9.4	2.6	0.23	0.44
SQLite1672	0.05%	0.98%	0.50%	6	3	3	0.1	0.1	0.0	0.06
Avg.	0.31%	3.08%	1.39%	-	-	-	-	-	-	-

Table 4: Overhead during regular execution and detailed performance comparison (red font denotes >3% overhead; #: count of dynamic instances; Abort%: percentage of aborted dynamic Txs.)

benchmarks in Table 3. It recovers the first 6 failures through HTM retries. It recovers from the Transmission failure through `longjmp` (it rolls back the `malloc` that cannot be handled by HTM-retry through `free`).

Unrecoverable benchmarks There are 9 benchmarks that no tools can help recover for mainly three reasons. Some of these issues go beyond the scope of failure recovery, yet others are promising to address in the future. First, two order violation benchmarks cause failures when the failure thread is unexpectedly slow. Therefore, re-executing the failure thread would not help correct the timing. Fortunately, both failures can be prevented by delaying resource deallocation, a prevention approach proposed before for memory-bug failures [29, 35]. Second, three benchmarks, Cherokee326, Apache25520, and MySQL169, cause failures that are difficult to detect (i.e., silent data corruption). Tackling them goes beyond the scope of failure recovery. Third, the remaining four failures cannot be recovered due to un-reexecutable instructions, which are promising to address. For example, Intel TSX does not support putting `memcpy`, `cond_wait`, or `I/O` into its Txs. More sophisticated TMs with OS support [37, 45] could help recover these failures.

7.2 Performance

Table 4 shows the regular-run overheads of applying BugTM schemes to 20 benchmarks, all the benchmarks that are recoverable by BugTM$_{HS}$.

BugTM$_H$ incurs more overhead, about 3% on average, than ConAir does, about 0.3% on average, mainly because a Tx is much more expensive than a `setjmp`.

Fortunately, BugTM$_{HS}$ wins most of the lost performance back, incurring 1.4% overhead on average and less than 3% for **all but 3** benchmarks. In the worst cases, it incurs 4.2% and 5.3% overhead for two benchmarks in Mozilla JavaScript Engine (JSE), a browser component with little I/O. If we apply BugTM$_{HS}$ to the whole browser, the overhead would be much smaller, as JSE never takes >20% of the whole page-loading time based on our profiling and previous work [31].

Comparing BugTM$_{HS}$ with BugTM$_H$, BugTM$_{HS}$ is faster mainly because it has greatly reduced the number of transactions at run time. For example, for the four benchmarks that incur the largest overhead under BugTM$_H$ (Moz-JS18025, Moz-JS142651, Click, and Moz-JS79054), BugTM$_{HS}$ reduces the #`StartTx` per 10μs from 9.4 — 30.4 to 2.6 — 12.6, and hence dropping the overhead from 8.11–11.9% to 2.6–5.3%.

Tx abort rate is less than 1% for all benchmarks, with more than 95% of all aborts being unknown aborts (timer interrupts, etc.). As Section 7.4 will show, abort rates and overhead are much worse in alternative designs.

Recovery time & Comparison with whole-program restart A successful BugTM failure recovery takes little time. In our experiments, the recovery of atomicity violations and deadlocks mostly takes less than 100 μ-seconds (median is 76 μ-seconds). The recovery of order violations takes slightly longer time, as it highly depends on how much `sleep` is inserted to trigger the failure. BugTM recovery is much faster than a system restart, which could take a few minutes or even more for complicated systems. It also avoids wasting already conducted computation and crash inconsistencies. For example, without BugTM, MySQL791 would crash the database after a table is changed but before this change is logged, leaving inconsistent persistent states.

Understanding BugTM$_H$ overhead The overhead of BugTM$_H$ differs among benchmarks, ranging from

	BugTM$_H$	Intra-proc	Trapping-Ins	Loop
Moz-xpcom	0.45% ✓	0.44% ✗	0.54% ✓	0.20% ✓
Moz-JS18025	9.03% ✓	7.01% ✓	16.8% ✓	11.3% ✓
Moz-JS79054	11.7% ✓	11.4% ✗	14.0% ✓	11.1% ✓
Moz-JS142651	11.9%✓	7.6% ✗	19.6% ✓	12.2%✓
MySQL791	1.98% ✓	1.50% ✓	11.4% ✓	11.5% ✓
MySQL2011	0.13% ✓	0.13% ✗	1.50% ✓	0.06% ✓
MySQL3596	3.10% ✓	3.05% ✓	108% ✗	2.63% ✓
MySQL16582	3.03% ✓	0.16% ✓	93.1% ✓	1.89% ✓
MySQL38883	3.08% ✓	3.04% ✓	106% ✗	2.52% ✓

Table 5: BugTM$_H$ vs. alternative designs (%: the overhead over baseline execution w/o recovery scheme applied; ✓: failure recovered; ✗: failure not recovered.)

0.00% to 11.9%. As TM researchers found before, performance in TM systems is often complicated [4, 34]. An indicating metrics for our benchmarks is the frequency of dynamic `StartTx`. As shown in the #`StartTx` per 10μs column of Table 4, BugTM$_H$ executes more than 1 `StartTx` per 10 micro second on average for 10 benchmarks, and incurs more than 1% overhead for 9 of them.

7.3 Diagnosis

BugTM$_{\text{HS}}$ can provide diagnosis information for all the 20 benchmarks that it can help recover from. For 13 benchmarks, recoveries through `longjmp` or HTM rollback are initiated right before explicit failures, for which BugTM$_{\text{HS}}$ provides accurate root-cause diagnosis following Figure 11. For the other 7, the recoveries are triggered by HTM data-conflict aborts, for which BugTM$_{\text{HS}}$ correctly suggests that there might be RAR, RAW, or WAR atomicity violations behind these aborts but cannot provide more detailed root-cause information.

BugTM$_{\text{HS}}$ provides the option to log memory accesses during failure recovery attempts initiated by `longjmp`. Evaluation shows that this extra logging incurs 1.01X – 2.5X slowdowns to failure recovery with no overhead to regular execution. The 2.5X slowdown happens during a fast half-microsecond recovery.

7.4 Alternative designs of BugTM

Table 5 shows the performance and recovery capability of three alternative designs of BugTM$_H$. Due to space constraints, we only show results on benchmarks in MySQL database server and Mozilla browser suite (non-extracted). Since BugTM$_H$ is the foundation of BugTM$_{\text{HS}}$, an alternative design that degrades the performance or recovery capability of BugTM$_H$ will also degrade BugTM$_{\text{HS}}$ accordingly as discussed below.

Inter-procedural vs. Intra-procedural BugTM$_H$ uses the inter-procedural algorithm discussed in Section 3.5. This design adds 0.00 – 4.3 % overhead to its intra-procedural alternative, as shown in Table 5. In exchange, there are 4 benchmarks in Table 5 that require inter-procedural re-execution of BugTM$_H$ to recover from. Among them, two can be recovered by ConAir and hence can still be recovered by intra-procedural BugTM$_{\text{HS}}$; the other two require inter-procedural BugTM$_{\text{HS}}$ to recover. Recovering MySQL2011, Moz-xpcom, Moz-JS79054 has to re-execute not only function F where failures occur, but also F's caller. As for Moz-JS142651, we need to re-execute a callee of F where a memory access involved in the atomicity violation resides.

Including trapping instructions in Txs Clearly, if BugTM$_H$ did not intentionally exclude system calls from its Txs, more Txs will abort. This alternative design hurts performance a lot, incurring around 100% overhead for three MySQL benchmarks shown in Table 5. Such design also causes BugTM$_{\text{HS}}$ to incur more than 20% overhead on these benchmarks. Furthermore, these aborts may hurt recovery capability, as they will cause corresponding Tx regions to execute in non-transaction mode to avoid endless aborts and hence lose the opportunity of failure recovery. This indeed happens for two benchmarks in Table 5. One of them will also fail to be recovered by BugTM$_{\text{HS}}$ under this alternative design.

Including loops in Txs could lead to more capacity aborts, which are indeed observed for all benchmarks in Table 5. The overhead actually does not change much for most benchmarks. Having said that, it raises the overhead of MySQL791 from 1.98% to 11.5%.

More Txs We also tried randomly inserting more `StartTx`. The overhead increases significantly. For Moz-JS142651, when we double, treble, and quadruple the number of dynamic Txs through randomly inserted Txs, the overhead goes beyond 30%, 100%, and 800%. The impact to BugTM$_{\text{HS}}$ would also be huge accordingly.

7.5 Discussion

As the evaluation and our earlier discussion show, BugTM does not guarantee to recover from all concurrency bug failures, particularly if the bug has a long error propagation before causing a failure. However, we believe BugTM, particularly BugTM$_{\text{HS}}$, would provide a beneficial safety net to most multi-threaded software with little deployment cost or performance loss.

Several practices can help further improve the benefit of BugTM. First, as discussed in Section 7.1, some improvements of HTM design would greatly help BugTM to recover from more concurrency-bug failures. Second, developers' practices of inserting sanity checks into software would greatly help BugTM. With more sanity checks, fewer concurrency bugs would have long error propagation and hence more concurrency-bug failures would be recovered by BugTM. Third, different from locks, which protect the atomicity of a code region only when the region and *all* its conflicting code are all protected by the same lock, BugTM can help protect a code region regardless how other code regions are written. Consequently, developers could choose to selectively apply BugTM to parts of software where he/she is least certain about synchronization correctness.

Finally, BugTM can be applied to software that is already using HTMs. BugTM will choose not to make its HTM regions nesting with existing HTM regions.

8 Related Work

Concurrency-bug failure prevention The prevention approach works by perturbing the execution timing, hoping that failure-triggering interleavings would not happen. It either relies on prior knowledge about a bug/failure [19, 27] to prevent the same bug from manifesting again, or relies on extensive off-line training [53, 51] to guide the production run towards likely failure-free timing. It is not suitable for avoiding production-run failures caused by previously unknown concurrency bugs. Particularly, the LiteTx work [51] proposes hardware extensions that are like lightweight HTM (i.e., without versioning or rollback) to constrain production-run thread interleavings, proactively prohibiting interleavings that have not been exercised during off-line testing. BugTM and LiteTx are fundamentally different on how they prevent/recover-from concurrency-bug failures and how they use hardware support.

Automated concurrency-bug fixing Static analysis and code transformation techniques have been proposed to automatically generate patches for concurrency bugs [17, 18, 25, 47]. They work at off-line and rely on accurate bug-detection results. A recent work [16] proposes a data-privatization technique to automatically avoid some read-after-write and read-after-read atomicity violations. When a thread may access the same shared variable with no blocking operations in between, this technique would create a temporary variable to buffer the result of the earlier access and feed it to the later read access. Although inspiring, this previous work is clearly different from BugTM. It does not handle many other types of concurrency bugs, including write-after-read and write-after-write atomicity violations and order violations. Furthermore, it relies on analyzing traces of previous execution of the program to carry out data privatization. The different usage contexts lead to different designs.

Failure recovery Rollback and re-execution have long been a valuable recovery [35, 44] and debugging [7, 20, 33, 43] technique. Many rollback-reexecution techniques target full system/application replay and hence are much more complicated and expensive than BugTM.

Feather-weight re-execution based on idempotency has been used before for recovering hardware faults [6, 9]. Using it to help recover from concurrency-bug failures was recently pioneered by ConAir [55]. BugTM greatly improved ConAir. BugTM$_H$ and ConAir use not only different rollback/reexecution mechanisms, but also completely different static analysis and code transformation. The `setjmp` and `longjmp` used by ConAir have different performance and correctness implications from `StartTx`, `CommitTx`, and `AbortTx`, which naturally led to completely different designs in BugTM$_H$ and ConAir.

Recent work leverages TM to help recover from transient hardware faults [21, 24, 49]. Due to the different types of faults/bugs these tools and BugTM are facing, their designs are different from BugTM. They wrap the whole program into transactions, which inevitably leads to large overhead (around 100% overhead [21, 49]) or lots of hardware changes to existing HTM [24], and different design about how/where to insert Tx APIs. They use different ways to detect and recover from the occurrence of faults, and hence have different Tx abort handling from BugTM. They either rely on **non**-existence of concurrency bugs to guarantee determinism [21] or only apply for single-threaded software [24, 49], which is completely different from BugTM.

Others Lots of research was done on HTM and STM [2, 3, 5, 11, 13, 14, 30, 36, 42]. Recent work explored using HTM to speed up distributed transaction systems [48], race detection [10, 54], etc. Previous empirical studies have examined the experience of using Txs, instead of locks, in developing parallel programs [38, 52]. They all look at different ways of using TM systems from BugTM.

9 Conclusions

Concurrency bugs severely affect system availability. This paper presents BugTM that leverages HTM available on commodity machines to help automatically recover concurrency-bug failures during production runs. BugTM can recover failures caused by all major types of concurrency bugs and incurs very low overhead (1.39%). BugTM does not require any prior knowledge about concurrency bugs in a program and guarantees not to introduce any new bugs. We believe BugTM improves the state of the art of failure recovery, presents novel ways of using HTM techniques, and provides a practical and easily deployable solution to improve the availability of multi-threaded systems with little cost.

10 Acknowledgments

We thank Wei Zhang and Linhai Song for their help in our experiments with ConAir. We are also grateful to the anonymous reviewers whose comments helped bring the paper to its final form. This project is funded by NSF (grants CNS-1563956, IIS-1546543, CNS-1514256, CCF-1514189, CCF-1439091) and CERES Center for Unstoppable Computing.

References

[1] Intel 64 and ia-32 architectures optimization reference manual. http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf. Accessed: 2016-07-30.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA*, 2005.

[3] Tongxin Bai, Xipeng Shen, Chengliang Zhang, William N. Scherer, Chen Ding, and Michael L. Scott. A key-based adaptive transactional memory executor. In *IPDPS*, 2007.

[4] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *ISCA*, 2007.

[5] Dhruva R. Chakrabarti, Prithviraj Banerjee, Hans-J. Boehm, Pramod G. Joisha, and Robert S. Schreiber. The runtime abort graph and its application to software transactional memory optimization. In *CGO*, 2011.

[6] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent processor architecture. In *MICRO '11*.

[7] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *OSDI*, 2014.

[8] Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum, and Marek Olszewski. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS*, 2009.

[9] Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott Mahlke, and David August. Encore: Low-cost, fine-grained transient fault recovery. In *MICRO '11*.

[10] Shantanu Gupta, Florin Sultan, Srihari Cadambi, Franjo Ivancic, and Martin Rotteler. Using hardware transactional memory for data race detection. In *IEEE IPDPS*, pages 1–11, 2009.

[11] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[12] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, et al. The ibm blue gene/q compute chip. *IEEE Micro*, 32(2):48–60, 2012.

[13] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2nd edition, 2010.

[14] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[15] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[16] Jeff Huang and Charles Zhang. Execution privatization for scheduler-oblivious concurrent programs. In *OOPSLA*, 2012.

[17] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[18] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *OSDI*, 2012.

[19] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI* https://code.google.com/archive/p/dimmunix/, 2008.

[20] Samuel King, George Dunlap, and Peter Chen. Debugging operating systems with time-traveling virtual machines. Proceedings of USENIX ATC, 2005.

[21] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Haft: hardware-assisted fault tolerance. In *EuroSys*, 2016.

[22] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[23] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[24] Jianli Li, Qingping Tan, and Lanfang Tan. En-HTM: Exploiting hardware transaction memory for achieving low-cost fault tolerance. In *2013 Fourth International Conference on Digital Manufacturing & Automation*.

[25] Peng Liu, Omer Tripp, and Charles Zhang. Grail: Context-aware fixing of concurrency bugs. In *FSE*, 2014.

[26] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

[27] Brandon Lucia and Luis Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *ASPLOS*, 2013.

[28] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA*, 2008.

[29] Vitaliy B. Lvin, Gene Novark, and Emery D. Berger. Archipelago: Trading address space for reliability and security. In *ASPLOS*, 2008.

[30] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *HPCA*, 2006.

[31] Javad Nejati and Aruna Balasubramanian. An in-depth study of mobile browser performance. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 1305–1315, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[32] PCWorld. Nasdaq's Facebook Glitch Came From Race Conditions. `http://www.pcworld.com/businesscenter/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html`.

[33] Gilles Pokam, Klaus Danne, Cristiano Pereira, Rolf Kassa, Tim Kranich, Shiliang Hu, Justin Emile Gottschlich, Nima Honarmand, Nathan Dautenhahn, Samuel T. King, and Josep Torrellas. Quickrec: prototyping an intel architecture extension for record and replay of multithreaded programs. In *ISCA*, 2013.

[34] Donald E. Porter and Emmett Witchel. Understanding transactional memory performance. In *ISPASS*, 2010.

[35] Feng Qin, Joe Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies — a safe method to survive software failure. In *SOSP*, 2005.

[36] R. Rajwar and J. R. Goodman. Transactional lock-free execution. In *ASPLOS*, 2002.

[37] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP*, 2007.

[38] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *PPoPP*, 2010.

[39] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *AADEBUG*, 2005.

[40] SecurityFocus. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

[41] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. Do I use the wrong definition? DefUse: Definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, 2010.

[42] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *ISCA*, 2007.

[43] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: Diagnosing production run failures at the user's site. In *SOSP*, 2007.

[44] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *SOSP*, 2011.

[45] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. xcalls: safe I/O in memory transactions. In *EuroSys*, 2009.

[46] Haris Volos, Andres Jaan Tack, Michael M. Swift, and Shan Lu. Applying transactional memory to concurrency bugs. In *ASPLOS*, 2012.

[47] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlk. Gadara: dynamic deadlock avoidance for mult-threaded programs. In *OSDI*, 2008.

[48] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *SOSP*, 2015.

[49] Gulay Yalcin, Osman S Unsal, Adrian Cristal, Ibrahim Hur, and Mateo Valero. Symptomtm: Symptom-based error detection and recovery using hardware transactional memory. In *IEEE PACT*, pages 199–200, 2011.

[50] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi N. Bairavasundaram. How do fixes become bugs? In *FSE*, 2011.

[51] Jie Yu and Satish Narayanasamy. Tolerating concurrency bugs using transactions as lifeguards. In *IEEE Micro*, pages 263–274, 2010.

[52] Jiaqi Zhang, Wenguang Chen, Xinmin Tian, and Weimin Zheng. Exploring the emerging applications for transactional memory. In *Proceedings of the 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2008.

[53] Mingxing Zhang, Yongwei Wu, Shan Lu, Shanxiang Qi, Jinglei Ren, and Weimin Zheng. AI: a lightweight system for tolerating concurrency bugs. In *FSE*, 2014.

[54] Tong Zhang, Dongyoon Lee, and Changhee Jung. Txrace: Efficient data race detection using commodity hardware transactional memory. In *ASPLOS*, 2016.

[55] Wei Zhang, Marc de Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution. In *ASPLOS*, 2013.

[56] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: Detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.

[57] Wei Zhang, Chong Sun, Junghee Lim, Shan Lu, and Thomas Reps. ConMem: Detecting Crash-Triggering Concurrency Bugs through an Effect-Oriented Approach. *ACM TOSEM*, 2012.

Tailwind: Fast and Atomic RDMA-based Replication

Yacine Taleb*, Ryan Stutsman†, Gabriel Antoniu*, Toni Cortes‡

**Univ Rennes, Inria, CNRS, IRISA †University of Utah, ‡BSC and UPC*

Abstract

Replication is essential for fault-tolerance. However, in in-memory systems, it is a source of high overhead. Remote direct memory access (RDMA) is attractive to create redundant copies of data, since it is low-latency and has no CPU overhead at the target. However, existing approaches still result in redundant data copying and active receivers. To ensure atomic data transfers, receivers check and apply only fully received messages. Tailwind is a zero-copy recovery-log replication protocol for scale-out in-memory databases. Tailwind is the first replication protocol that eliminates *all* CPU-driven data copying and fully bypasses target server CPUs, thus leaving backups idle. Tailwind ensures all writes are atomic by leveraging a protocol that detects incomplete RDMA transfers. Tailwind substantially improves replication throughput and response latency compared with conventional RPC-based replication. In symmetric systems where servers both serve requests and act as replicas, Tailwind also improves normal-case throughput by freeing server CPU resources for request processing. We implemented and evaluated Tailwind on RAMCloud, a low-latency in-memory storage system. Experiments show Tailwind improves RAMCloud's normal-case request processing throughput by 1.7×. It also cuts down writes median and 99th percentile latencies by 2x and 3x respectively.

1 Introduction

In-memory key-value stores are an essential building block for large-scale data-intensive applications [3, 19]. Recent research has led to in-memory key-value stores that can perform millions of operations per second per machine with a few microseconds remote access times. Harvesting CPU power and eliminating conventional network overheads has been key to these gains. However, like many other systems, they must replicate data in order to survive failures.

As the core frequency scaling and multi-core architecture scaling are both slowing down, it becomes critical to reduce replication overheads to keep-up with shifting application workloads in key-value stores [13]. We show that replication can consume up to 80% of the CPU cycles for write-intensive workloads (§4.4), in strongly-consistent in-memory key-value stores. Techniques like remote-direct memory access (RDMA) are promising to improve overall CPU efficiency of replication and keep predictable tail latencies.

Existing RDMA-based approaches use message-passing interfaces: a sender remotely places a message into a receiver's DRAM; a receiver must actively poll and handle new RDMA messages. This approach guarantees the atomicity of RDMA transfers, since only fully received messages are applied by the receiver [4, 10, 30]. However, this approach defeats RDMA efficiency goals since it forces receivers to use their CPU to handle incoming RDMA messages and it incurs additional memory copies.

The main challenge of efficiently using RDMA for replication is that failures could result in partially applied writes. The reason is that receivers are not aware of data being written to their DRAM. Leaving receivers idle is challenging because there is no protocol to guarantee data consistency in the event of failures.

A second key limitation with RDMA is its low scalability. This limitation comes from the connection-oriented nature of RDMA transfers. Senders and receivers have to setup queue pairs (QP) to perform RDMA. Lots of recent work has observed the high cost of NIC connection cache misses [4, 11, 32]. Scalability is limited as it typically depends on the cluster size.

To address the above challenges, we developed Tailwind, a zero-copy primary-backup log replication protocol that completely bypasses CPUs on all target backup servers. In Tailwind, log records are transferred directly from the source server's DRAM to I/O buffers at target servers via RDMA writes. Backup servers are *completely passive* during replication, saving their CPUs for other purposes; they flush these buffers to solid-state drives (SSD) periodically when the source triggers it via remote procedure call (RPC) or when power is interrupted. Even though backups are idle during replication, Tailwind is strongly consistent: it has a protocol that allows backups to detect incomplete RDMA transfers.

Tailwind uses RDMA write operations for all data movement, but all control operations such as buffer allocation and setup, server failure notifications, buffer flushing and freeing are all handled through conventional RPCs. This simplifies such complex operations without slowing down data movement. In our implementation, RPCs only account for 10^{-5} of the replication requests. This also makes Tailwind easier to use in systems that use log replication over distributed blocks even if they were not designed to exploit RDMA.

Since Tailwind needs only to maintain connections between a primary server and its backups, the number of connections scales with the size of a replica group, not with the cluster size, making Tailwind a scalable approach.

We implemented and evaluated Tailwind on RAMCloud, a scale-out in-memory key-value store that exploits fast kernel-bypass networking. Tailwind is suited to RAMCloud's focus on strong consistency and low latency. Tailwind significantly improves RAMCloud's throughput since each PUT operation in the cluster results in three remote replication operation that would otherwise consume server CPU resources.

Tailwind improves RAMCloud's throughput by 1.7× on the YCSB benchmark, and it reduces durable PUT median latency from 32 µs to 16 µs and 99th percentile latency from 78 µs to 28 µs. Theses results stem from the fact that Tailwind significantly reduces the CPU cycles used by the replication operations: Tailwind only needs 1/3 of the cores RAMCloud uses to achieve the same throughput.

This paper makes four key contributions;

- it analyzes and quantifies CPU related limitations in modern in-memory key-value stores;
- it presents Tailwind's design, it describes its implementation in the RAMCloud distributed in-memory key-value store, and it evaluates its impact on RAMCloud's normal-case and recovery performance;
- to our knowledge, Tailwind is the first log replication protocol that eliminates all superfluous data copying between the primary replica and its backups, and it is the first log replication protocol that leaves servers CPU idle while serving as replication targets; this allows servers to focus more resources on normal-case request processing;
- Tailwind separates the replication data path and control path and optimizes them individually; it uses RDMA for heavy transfer, but it retains the simplicity of RPC for rare operations that must deal with complex semantics like failure handling and resource exhaustion.

2 Motivation and Background

Replication and redundancy are fundamental to fault tolerance, but at the same time they are costly. Primary-backup replication (PBR) is popular in fault-tolerant storage systems like file systems and key-value stores, since it tolerates f stop-failures with $f+1$ replicas. Note that, we refer to a primary replica server as *primary*, and secondary replica server as *secondary* or *backup*. In some systems, backup servers don't process user-facing requests, but in many systems each node acts as both a primary for some data items and as a backup for other data items. In some systems this is implicit: for example, a key-value store may store its state on HDFS [28], and a single physical machine might run both a key-value store frontend and an HDFS chunkserver.

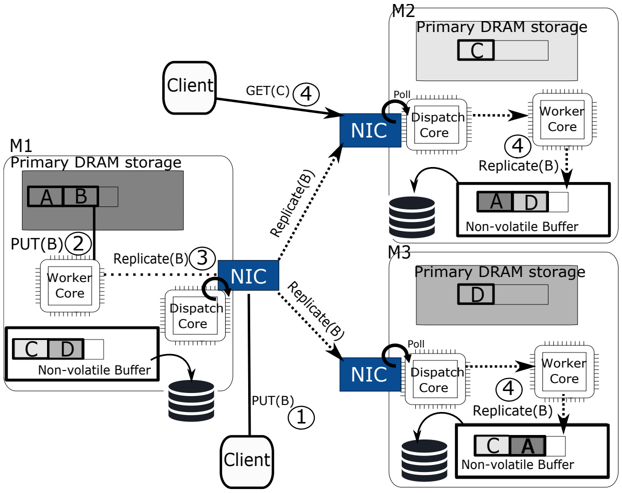


Figure 1: Flow of primary-backup replication

Replication is expensive for three reasons. First, it is inherently redundant and, hence, brings overhead: the act of replication itself requires moving data over the network. Second, replication in strongly consistent systems is usually synchronous, so a primary must stall while holding resources while waiting for acknowledgements from backups (often spinning a CPU core in low-latency stores). Third, in systems, where servers (either explicitly or implicitly) serve both client-facing requests and replication operations, those operations contend.

Figure 1 shows this in more detail. Low-latency, high-throughput stores use kernel-bypass to directly poll NIC control (with a dispatch core) rings to avoid kernel code paths and interrupt latency and throughput costs. Even so, a CPU on a primary node processing an update operation must receive the request, hand the request off to a core (worker core) to be processed, send remote messages, and then wait for multiple nodes acting as backup to process these requests. Batching can improve the number of backup request messages each server must receive, but at the cost of increased latency. Inherently, though, replication can double, triple, or quadruple the number of messages and the amount of data generated by client-issued write requests. It also causes expensive stalls at the primary while it waits for responses. In these systems, responses take a few microseconds which is too short a time for the primary to context switch to another thread, yet its long enough that the worker core spends a large fraction of its time waiting.

2.1 The Promise of RDMA

Recently, remote-direct memory access (RDMA) has been used in several systems to avoid kernel overhead and to reduce CPU load. Though the above kernel-bypass-based request processing is sometimes called (*two-sided*) RDMA, it still incurs request dispatching

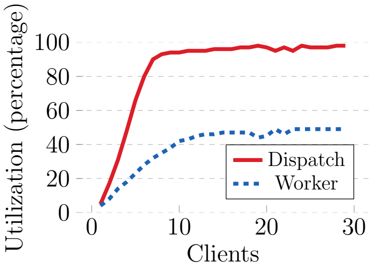


Figure 2: Dispatch and worker cores utilization percentage of a single RAMCloud server. Requests consist of 95/5 GET/PUT ratio.

and processing overhead because a CPU, on the destination node, must poll for the message and process it. RDMA-capable NICs also support so called *one-sided* RDMA operations that directly access the remote host's memory, bypassing its CPU altogether. Remote NICs directly service RDMA operations without interrupting the CPU (neither via explicit interrupt nor by enqueuing an operation that the remote CPU must process). One-sided operations are only possible through reliable-connected queue pairs (QP) that ensure in-order and reliable message delivery, similar to the guarantees TCP provides.

2.1.1 Opportunities

One-sided RDMA operations are attractive for replication; replication inherently requires expensive, redundant data movement. Backups are (mostly) passive; they often act as dumb storage, so they may not need CPU involvement. Figure 2 shows that RAMCloud, an in-memory low-latency kernel-bypass-based key-value store, is often bottlenecked on CPU (see §4 for experimental settings). For read-heavy workloads, the cost of polling network and dispatching requests to idle worker cores dominates. Only 8 clients are enough to saturate a single server dispatch core. Because of that, worker cores cannot be fully utilized. One-sided operations for replicating PUT operations would reduce the number of requests each server handles in RAMCloud, which would indirectly but significantly improve read throughput. For workloads with a significant fraction of writes or where a large amount of data is transferred, write throughput can be improved, since remote CPUs needn't copy data between NIC receive buffers and I/O or non-volatile storage buffers.

2.1.2 Challenges

The key challenge in using one-sided RDMA operations is that they have simple semantics which offer little control on the remote side. This is by design; the remote NIC executes RDMA operations directly, so they lack the generality that a conventional CPU-based RPC handlers would have. A host can issue a remote *read* of a single, sequential region of the remote processes virtual address space (the region to read must be *registered* first, but a process could register its whole virtual address space). Or, a host can issue a remote *write* of a single, sequential region of the remote processes virtual address space (again, the region must be registered with the NIC). NICs support a few more complex operations (compare-and-swap, atomic add), but these operations are currently much slower than issuing an equivalent two-sided operation that is serviced by the remote CPU [11, 30]. These simple, restricted semantics make RDMA operations efficient, but they also make them hard to use safely and correctly. Some existing systems use one-sided RDMA operations for replication (and some also even use them for normal case operations [4, 5]).

However, no existing primary-backup replication scheme reaps the full benefits of one-sided operations. In existing approaches, source nodes send replication operations using RDMA writes to push data into ring buffers. CPUs at backups poll for these operations and apply them to replicas. In practice, this is is effectively emulating two-sided operations [4]. RDMA reads don't work well for replication, because they would require backup CPUs to schedule operations and "pull" data, and primaries wouldn't immediately know when data was safely replicated.

Two key, interrelated issues make it hard to use RDMA writes for replication that fully avoids the remote CPUs at backups. First, a primary can crash when replicating data to a backup. Because RDMA writes (inherently) don't buffer all of the data to be written to remote memory, its possible that an RDMA write could be partially applied when the primary crashes. If a primary crashes while updating state on the backup, the backup's replica wouldn't be in the "before" or "after" state, which could result in a corrupted replica. Worse, since the primary was likely mutating all replicas concurrently, it is possible for all replicas to be corrupted. Interestingly, backup crashes during RDMA writes don't create new challenges for replication, since protocols must deal with that case with conventional two-sided operations too. Well-known techniques like log-structured backups [18, 23, 25] or shadow paging [35] can be used to prevent update-in-place and loss of atomicity. Traditional log implementations enforce a total ordering of log entries [9]. In database systems, for instance, the order is used to recreate a consistent state during recovery.

Unfortunately, a second key issue with RDMA operations makes this hard: each operation can only affect a single, contiguous region of remote memory. To be efficient, one-sided writes must replicate data in its final, stable form, otherwise backup CPU must be involved, which defeats the purpose. For stable storage, this generally requires some metadata. For example, when a backup uses data found in memory or storage it must know which portions of memory contain valid objects, and it must be able to verify that the objects and the markers that delineate them haven't been corrupted. As a result, backups need some metadata about the objects that they host in addition to the data items themselves. However, RDMA writes make this hard. Metadata must inherently be intermixed with data objects, since RDMA writes are contiguous. Otherwise, multiple round trips would be needed, again defeating the efficiency gains.

Tailwind solves these challenges through a form of low-overhead redundancy in log metadata. Primaries incrementally log data items and metadata updates to remote memory on backups via RDMA writes. Backups remain unaware of the contents of the buffers and blindly flush them to storage. In the rare event when a primary fails, all backups work in parallel scanning log data to reconstruct metadata so data integrity can be checked. The next section describes its design in detail.

3 Design

Tailwind is a strongly-consistent RDMA-based replication protocol. It was designed to meet four requirements:

Zero-copy, Zero-CPU on Backups for Data Path. In order to relieve backups CPUs from processing replication requests, Tailwind relies on one-sided RDMA writes for all data movement. In addition, it is zero-copy at primary and secondary replicas; the sender uses kernel-bypass and scatter/gather DMA for data transfer; on the backup side, data is directly placed to its final storage location via DMA transfer without CPU involvement.

Strong Consistency. For every object write Tailwind synchronously waits for its replication on all backups before notifying the client. Although RDMA writes are one-sided, reliable-connected QPs generate work completion to notify the sender once a message has been correctly sent and acknowledged by the receiver NIC (i.e. written to remote memory) [8]. One-sided operation raise many issues, Tailwind is designed to cover *all* corner cases that may challenge correctness (§3.4).

Overhead-free Fault-Tolerance. Backups are unaware of replication as it happens, which can be unsafe in case of failures. To address this, Tailwind appends a piece of metadata in the log after every object update. Backups use this metadata to check integrity and locate valid objects during recovery. Although a few backups have to do little extra work during crash recovery, that work has no impact on recovery performance (§4.6).

Preserves Client-facing RPC Interface. Tailwind has no requirement on the client side; all logic is implemented between primaries and backups. Clients observe the same consistency guarantees. However, for write operations, Tailwind highly improves end-to-end latency and throughput from the client perspective (§4.2).

3.1 The Metadata Challenge

Metadata is crucial for backups to be able to use replicated data. For instance, a backup needs to know which portions of the log contain valid data. In RPC-based systems, metadata is usually piggybacked within a replication request [11, 21]. However, it is challenging to update both data and metadata with a single RDMA write since it can only affect a contiguous memory region. In this case, updating both data and metadata would require sending two messages which would nullify one-sided RDMA benefits. Moreover, this is risky: in case of failures a primary may start updating the metadata and fail before finishing, thereby invalidating all replicated objects.

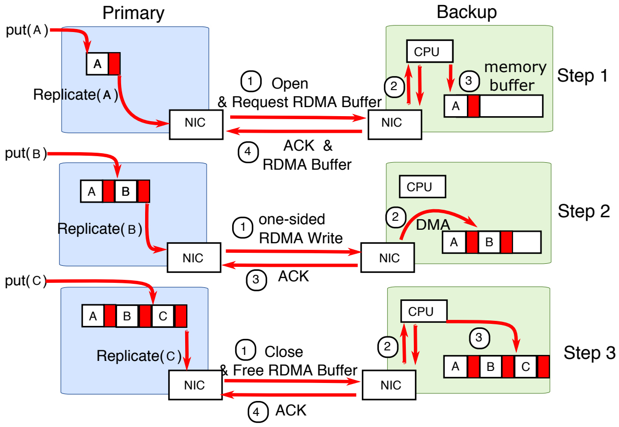


Figure 3: The three replication steps in Tailwind. During the first (open) and third (close) steps, the communication is done through RPCs. While the second step involves one-sided writes only.

For log-structured data, backups need two pieces of information: (1) the *offset* through which data in the buffer is valid. This is needed to guarantee the atomicity of each update. An outdated offset may lead the backup to use old and inconsistent data during crash recovery. (2) A *checksum* used to check the integrity of the length fields of each log record during recovery. Checksums are critical for ensuring log entry headers are not corrupted while in buffers or on storage. These checksums ensure iterating over the buffer is safe; that is, a corrupted length field does not “point” into the middle of another object, out of buffer, or indicate an early end to the buffer.

The protocol assumes that each object has a *header* next to it [4, 12, 26]. Implementation-agnostic information in headers should include: (1) the size of the object next to it to allow log traversal; (2) an integrity check that ensures the integrity of the contents of the log entry.

Tailwind checksums are 32-bit CRCs computed over log entry headers. The last checksum in the buffer covers all previous headers in the buffer. For maximum protection, checksums are end-to-end: they should cover the data while it is in transit over the network and while it occupies storage.

To be able to perform atomic updates with one-sided RDMAs in backups, the last checksum and the current offset in the buffer must be present and consistent in the backup after *every* update. A simple solution is to append the checksum and the offset before or after every object update. A single RDMA write would suffice then for both data and metadata. The checksum *must* necessarily be sent to the backup. Interestingly, this is not the case for the offset. The nature of log-structured data and the properties of one-sided RDMA make it possible, with careful design, for the backup to compute this value at recovery time without hurting consistency. This is possible because RDMA writes are performed (at the receiver side) in an increasing address order [8]. In addition, reliable-connected QPs ensure that updates are applied in the order they were sent.

Based on these observations, Tailwind appends a checksum in the log after every object update; at any point of time a checksum guarantees, with high probability, the integrity of all previous headers preceding it in the buffer. During failure-free time, a backup is ensured to always have the latest checksum, at the end of the log. On the other hand, backups have to compute the offset themselves during crash recovery.

3.2 Non-volatile Buffers

In Tailwind, at start up, each backup machine allocates a pool of in-memory I/O buffers (8 MB each, by default) and registers them with the NIC. To guarantee safety, each backup limits the number of buffers outstanding unflushed buffers it allows. This limit is a function of its local, durable storage speed. A backup denies opening a new replication buffer for a primary if it would exceed the amount of data it could flush safely to storage on backup power. Buffers are pre-zeroed. Servers require power supplies that allow buffers to be flushed to stable storage in the case of a power failure [4, 5, 20]. This avoids the cost of a synchronous disk write on the fast path of PUT operations.

Initiatives such as the OpenCompute Project propose standards where racks are backed by batteries backup, that could provide a minimum of 45 seconds of power supply [1] at full load, including network switches. Battery-backed DIMMs could have been another option, but they require more careful attention. Because we use RDMA, batteries need to back the CPU, the PCIe controller, and the memory itself. Moreover, there exists no clear way to flush data that could still be residing in NIC cache or in PCIe controller, which would lead to firmware modifications and to a non-portable solution.

3.3 Replication Protocol

3.3.1 Write Path

To be able to perform replication through RDMA, a primary has to has to reserve an RDMA-registered memory buffer from a secondary replica. The first step in Figure 3 depicts this operation: a primary sends an `open` RPC to a backup ①. Tailwind does not enforce any replica placement policy, instead it leaves backup selection up to the storage system. Once the `open` processed ② + ③, the backup sends an acknowledgement to the primary and piggybacks necessary information to perform RDMA writes ④ (i.e. `remote_key` and `remote_address` [8]). The open call fails if there are no available buffers. The primary has then to retry.

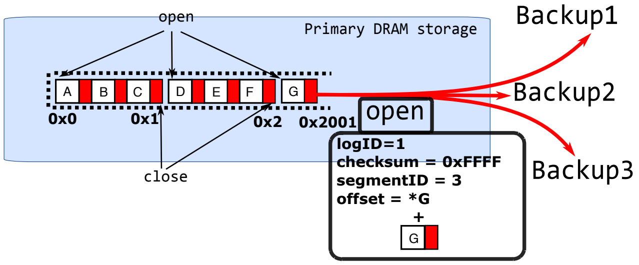


Figure 4: Primary DRAM storage consists of a monotonically growing log. It is logically split into fixed-size segments.

At the second step in Figure 3, the primary is able to perform all subsequent replication requests with RDMA writes ①. Backup NIC directly put objects to memory buffers via DMA transfer ② without involving the CPU. The primary gets work completion notification from its corresponding QP ③.

The primary keeps track of the last written offset in the backup buffer. When the next object would exceed the buffer capacity, the primary proceeds to the third step in Figure 3. The last replication request is called `close` and is performed through an RPC ①. The `close` notifies the backup ② + ③ that the buffer is full and thus can be flushed to durable storage. This eventually allows Tailwind to reclaim buffers and make them available to other primaries. Buffers are zeroed again when flushed.

We use RPCs for `open` and `close` operations because it simplifies the design of the protocol without hurting latency. As an example of complication, a primary may choose a secondary that has no buffers left. This can be challenging to handle with RDMA. Moreover, these operations are not frequent. If we consider 8 MB buffers and objects of 100 B, which corresponds to real workloads object size [19], `open` and `close` RPCs would account for 2.38×10^{-5} of the replication requests. Larger buffers imply less RPCs but longer times to flush backup data to secondary storage.

Thanks to all these steps, Tailwind can effectively replicate data using one-sided RDMA. However, without taking care of failure scenarios the protocol would not be correct. Next, we define essential notions Tailwind relies on for its correctness.

3.3.2 Primary Memory Storage

The primary DRAM log-based storage is logically sliced into equal *segments* (Figure 4). For every `open` and `close` RPC the primary sends a metadata information about current state: *logID*, latest checksum, *segmentID*, and current offset in the last segment. In case of failures, this information helps the backup in finding backup-data it stores for the crashed server.

At any given time, a primary can only replicate a single segment to its corresponding backups. This means a backup has to do very little work during recovery; if a primary replicates to *r* replicas then only *r* segments would be open, in case the primary fails.

```
input  : Pointer to a memory buffer rdmaBuf
output: Size of durably replicated data offset
currPosition = rdmaBuf ;
offset = currPosition ;
while currPosition < MAX_BUFFER_SIZE do
    /* Create a header in the current position */
    header = (Header)currPosition;
    currPosition += sizeof(header);
    /* Not Corrupted or incomplete header */
    if header→type != INVALID then
        if header→type == checksumType then
            checksum = (Checksum)currPosition;
            if checksum != currChecksum then
                /* Primaries never append a zero
                   checksum, check if it is 1. */
                if currChecksum == 0 and checksum == 1 then
                    offset = currPosition + sizeOf(checksum);
                else
                    return offset;
            else
                /* Move the offset at the end of
                   current checksum */
                offset = currPosition + sizeOf(checksum);
        else
            currChecksum = crc32(currChecksum, header);
    else
        return offset;
    /* Move forward to next entry */
    currPosition += header→objectSize;
/* We should only reach this line if a primary
   crashed before sending close */
return offset;
```

Algorithm 1: Updating RDMA buffer metadata

3.4 Failure Scenarios

When a primary or secondary replica fail the protocol must recover from the failure and rebuild lost data. Primary and secondary replicas failure scenarios require different actions to recover.

3.4.1 Primary-replica Failure

Primary replica crashes are one of the major concerns in the design. In case of such failures, Tailwind has to: (1) locate backup-data (of crashed primary) on secondary replicas; (2) rebuild up-to-date metadata information on secondary replicas; (3) ensure backup-data integrity and consistency; (4) start recovery.

Locating Secondary Replicas. After detecting a primary replica crash, Tailwind sends a query to all secondary replicas to identify the ones storing data belonging to the crashed primary. Since every primary has a unique logID it is easy for backups to identify which buffers belong to that logID.

Building Up-to-date Metadata. Backup buffers can either be in `open` or `close` states. Buffers that are `closed` do not pose any issue, they already have up-to-date metadata. If they are in disk or SSD they will be loaded to memory to get ready for recovery. However, for `open` buffers, the backup has to compute the offset and find the last checksum. Secondary replicas have to **scan** their `open` buffers to update their respective checksum and offset. To do so, they iterate over all entries as depicted in Algorithm 1.

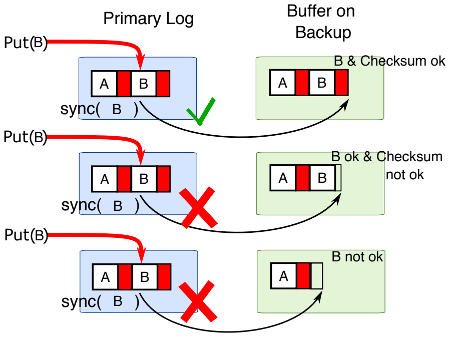


Figure 5: From top to bottom are three scenarios that can happen when a primary replica crashes while writing an object (B in this case) then synchronizing with backups. In the first scenario the primary replica fully writes the message to the backup leaving the backup in a correct state. B can be recovered in this case. In the second scenario, the object B is written but the checksum is partially written. Therefore, B is discarded. Similarly for the third scenario where B is partially written.

Basically, the algorithm takes an `open` buffer and tries to iterate over its entries. It moves forward thanks to the size of the entry which should be in the header. For every entry the backup computes a checksum over the header. When reaching a checksum in the buffer it is compared with the most recently computed checksum: the algorithm stops in case of checksum mismatch. There are three stop conditions: (1) integrity check failure; (2) invalid object found; (3) end of buffer reached.

A combination of three factors guarantee the correctness of the algorithm: (1) **the last entry is always a checksum**; Tailwind implicitly uses this condition as an end-of-transmission marker. (2) **Checksums are not allowed to be zero**; the primary replica always verifies the checksum before appending it. If it is zero it sets it to 1 and then appends it to the log. Otherwise, an incomplete object could be interpreted as valid zero checksum. (3) **Buffers are pre-zeroed**; combined with condition (2), a backup has a means to correctly detect the last valid offset in a buffer by using Algorithm 1.

3.4.2 Corrupted and Incomplete Objects

Figure 5 shows the three states of a backup RDMA buffer in case a primary replica failure. The first scenario shows a successful transmission of an object B and the checksum ahead of it. If the primary crashes, the backup is able to safely recover all data (i.e. A and B).

In the second scenario. the backup received B, but the checksum was not completely received. In this case the integrity check will fail. Object A will be recovered and B will be ignored. This is safe, since the client's PUT of B could not have been acknowledged.

The third scenario is similar: B was not completely transmitted to the backup. However, there creates two possibilities. If B's header was fully transmitted, then the algorithm will look past the entry and find a 0-byte at the end of the log entry. This way it can tell that the RDMA operation didn't complete in full, so it will discard the entry and treat the prefix of the buffer up through A as valid. If the checksum is partially written, it will still be discarded, since it will necessarily end in a 0-byte: something that is disallowed for all valid checksums that the primary creates. If B's header was only partially written, some of the bytes of the length field may be left zero. Imagine that o is the offset of the start of B. If the primary intended B to have length l and l' is the length actually written into the backup buffer. It is necessarily the case that $l' < l$, since lengths are unsigned and l' is a subset of the bits in l. As a result, $o + l'$ falls within the range where the original object data should have been replicated in the buffer. Furthermore, the data there consists entirely of zeroes, since an unsuccessful RDMA write halts replication to the buffer, and replication already halted before $o +$ `sizeof(Header)`. As a result, this case is handled identically to the incomplete checksum case, leaving the (unacknowledged) B off of the valid prefix of the buffer.

A key property maintained by Tailwind is that torn writes never depend on checksum checks for correctness. They can also be detected by careful construction of the log entries headers and checksums and the ordering guarantees that RDMA NICs provide.

Bit-flips The checksums, both covering the log entry headers and the individual objects themselves ensure that recovery is robust against bit-flips. The checksums ensure with high probability that bit-flip anywhere in any replica will be detected. In `closed` segments, whenever data corruption is detected, Tailwind declares the replica corrupted. The higher-level system will still successfully recover a failed primary, but it must rely on replicas from other backups to do so. In `open` segments data corruption is treated as partially transmitted buffers; as soon as Tailwind immediately stops iterating over the buffer and returns the last valid offset.

3.4.3 Secondary-replica Failure

When a server crashes the replicas it contained become unavailable. Tailwind must re-create new replicas on other backups in order to keep the same level of durability. Luckily, secondary-replica crashes are dealt with naturally in storage systems and do not suffer from one-sided RDMA complications. Tailwind takes several steps to allocate a new replica: (1) It suspends all operations on the corresponding primary replica; (2) It **atomically** creates a new secondary replica; (3) It resumes normal operations on the primary replica. Step (1) ensures that data will always have the same level of durability. Step (2) is crucial to avoid inconsistencies if a primary crashes while re-creating a secondary replica. In this case the newly created secondary replica would not have all data and cannot be used.

However, it can happen that a secondary replica stops and restarts after some time, which could lead to inconsistent states between secondary replicas. To cope with this, Tailwind keeps, at any point of time, a version number for replicas. If a secondary replica crashes, Tailwind updates the version number on the primary and secondaries. Since secondaries need to be notified, Tailwind uses an RPC instead of an RDMA for this operation. Tailwind updates the version number right after the step (2) when re-creating a secondary replica. This ensures that the primary and backups are synchronized. Replication can start again from a consistent state. Note that this RPC is rare and only occurs after the crash of a backup.

3.4.4 Network Partitioning

It can happen that a primary is considered crashed by a subset of servers. Tailwind would start locating its backups, then rebuilding metadata on those backups. While metadata is rebuilt, the primary could still perform one-sided RDMA to its backups, since they are always unaware of these type of operations. To remedy this, as soon as a primary or secondary failure is detected, all machines close their respective QPs with the crashed server. This allows Tailwind to ensure that servers that are alive but considered crashed by the environment do not interfere with work done during recovery.

4 Evaluation

We implemented Tailwind on RAMCloud a low-latency in-memory key-value store. Tailwind's design perfectly suits RAMCloud in many aspects:

Low latency. RAMCloud's main objective is to provide low-latency access to data. It relies on fast networking and kernel-bypass to provide a fast RPC layer. Tailwind can further improve RAMCloud (PUT) latency (§4.2) by employing one-sided RDMA without any additional complexity or resource usage.

Replication and Threading in RAMCloud. To achieve low latency, RAMCloud dedicates one core solely to poll network requests and dispatch them to worker cores (Figure 1). Worker cores execute all client and system tasks. They are never preempted to avoid context switches that may hurt latency. To provide strong consistency, RAMCloud always requests acknowledgements from all backups for every update. With the above threading-model, replication considerably slows down the overall performance of RAMCloud [31]. Hence Tailwind can greatly improve RAMCloud's CPU-efficiency and remove replication overheads.

Log-structured Memory. RAMCloud organizes its memory as an append-only log. Memory is sliced into smaller chunks called *segments* that also act as the unit of replication, i.e., for every segment a primary has to choose a backup. Such an abstraction makes it easy to replace RAMCloud's replication system with Tailwind. Tailwind checksums can be appended in the log-storage, with data, and replicated with minimal changes to the code. In addition, RAMCloud provides a log-cleaning mechanism which can efficiently clean old checksums and reclaim their storage space.

CPU	Xeon E5-2450 2.1 GHz 8 cores, 16 hw threads
RAM	16 GB 1600 MHz DDR3
NIC	Mellanox MX354A CX3 @ 56 Gbps
Switch	36 port Mellanox SX6036G
OS	Ubuntu 15.04, Linux 3.19.0-16, MLX4 3.4.0, libibverbs 1.2.1

Table 1: Experimental cluster configuration.

We compared Tailwind with RAMCloud replication protocol, focusing our analysis on three key questions:

Does Tailwind improve performance? Measurements show Tailwind reduces RAMCloud's median write latency by 2× and 99th percentile latency by 3× (Figure 7). Tailwind improves throughput by 70% for write-heavy workloads and by 27% for workloads that include just a small fraction of writes.

Why does Tailwind improve performance? Tailwind improves per-server throughput by eliminating backup request processing (Figure 9), which allows servers to focus effort on processing user-facing requests.

What is the Overhead of Tailwind? We show that Tailwind's performance improvement comes at no cost. Specifically, we measure and find no overhead during crash recovery compared to RAMCloud.

4.1 Experimental Setup

Experiments were done on a 35 server Dell r320 cluster (Table 1) on the CloudLab [24] testbed.

We used three YCSB [2] workloads to evaluate Tailwind: update-heavy (50% PUTs, 50% GETs), read-heavy (5% PUTs, 95% GETs), and update-only (100% PUTs). We intitially inserted 20 million objects of 100 B plus 30 B for the key. Afterwards, we ran up to 30 client machines. Clients generated requests according to a Zipfian distribution ($\theta = 0.99$). Objects were uniformly inserted in active servers. The replication factor was set to 3 and RDMA buffers size was set to 8 MB. Every data point in the experiments is averaged over 3 runs.

RAMCloud's RPC-based replication protocol served as a baseline for comparison. Note that, in the comparison with Tailwind, we refer to RAMCloud's replication protocol as RAMCloud for simplicity.

4.2 Performance Improvement

The primary goal of Tailwind is to accelerate basic operations throughput and latency. To demonstrate how Tailwind improves performance we show Figure 6, i.e. throughput per server as we increase the number of clients. When client operations consist of 5% PUTs and 95% GETs, RAMCloud achieves 500 KOp/s per server while Tailwind reaches up to 635 KOp/s. Increasing the update load enables Tailwind to further improve the throughput. For instance with 50% PUTs Tailwind sustains 340 KOp/s against 200 KOp/s for RAMCloud, which is a 70% improvement. With update-only workload, improvement is not further increased: In this case Tailwind improves the throughput by 65%.

Tailwind can improve the number of read operations serviced by accelerating updates. CPU cycles saved allow servers (that are backups as well) to service more requests in general.

Figure 7 shows that update latency is also considerably improved by Tailwind. Under light load Tailwind reduces median and 99th percentile latency of an update from 16 µs to 11.8 µs and from 27 µs to 14 µs respectively. Under heavy load, i.e. 500 KOp/s Tailwind reduces median latency from 32 µs to 16 µs compared to RAMCloud. Under the same load tail latency is even further reduced from 78 µs to 28 µs.

Tailwind can effectively reduce end-to-end client latency. With reduced acknowledgements waiting time, and more CPU power to process requests faster, servers can sustain a very low latency even under heavy concurrent accesses.

4.3 Gains as Backup Load Varies

Since all servers in RAMCloud act as both backups and primaries, Tailwind accelerates normal-case request processing indirectly by eliminating the need for servers to actively process replication operations. Figure 8 shows the impact of this effect. In each trial load is directed at a subset of four RAMCloud storage nodes; "Active Primary Servers" indicates the number of storage nodes that process client requests. Nodes do not replicate data to themselves, so when only one primary is active it is receiving no backup operations. All of the active primary's backup operations are directed to the other three otherwise idle nodes. Note that, in this figure, throughput is per-active-primaries. So, as more primaries are added, the aggregate cluster throughput increases.

As client GET/PUT operations are directed to more nodes (more active primaries), each node slows down because it must serve a mix of client operations intermixed with an increasing number of incoming backup operations. Enough client load is offered (30 clients) so that storage nodes are the bottleneck at all points in the graphs. With four active primaries, every server node is saturated processing client requests and backup operations for all client-issued writes.

Even when only 5% of client-issued operations are writes (Figure 8a), Tailwind increasingly improves performance as backup load on nodes increases. When a primary doesn't perform backup operations Tailwind improves throughput 3%, but that increases to 27% when the primary services its fair share of backup operations. The situation is similar when client operations are a 50/50 mix of reads and writes (Figure 8b) and when clients only issue writes (Figure 8c).

As expected, Tailwind enables increasing gains over RAMCloud with increasing load, since RDMA elimi-

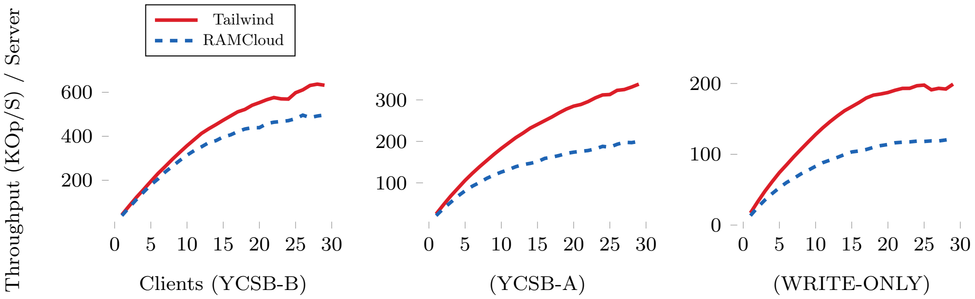
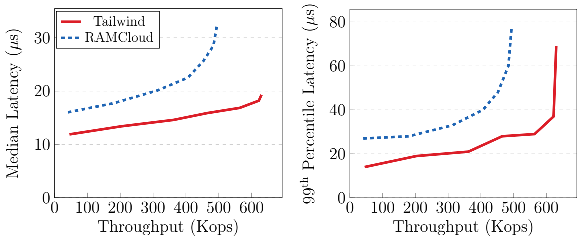


Figure 6: Throughput per server in a 4 server cluster.



(a) (b)

Figure 7: (a) Median latency and (b) 99th percentile latency of PUT operations when varying the load.

nates three RPCs that each server must handle for each client-issued write, which, in turn, eliminates worker core stalls on the node handling the write.

In short, the ability of Tailwind to eliminate replication work on backups translates into more availability for normal request processing, and, hence, better GET/PUT performance.

4.4 Resource Utilization

The improvements above have shown how Tailwind can improve RAMCloud's baseline replication normal-case. The main reason is that operations contend with backup operations for worker cores to process them. Figure 9a illustrates this: we vary the offered load (updates-only) to a 4-server cluster and report aggregated active worker cores. For example, to service 450 KOp/s, Tailwind uses 5.7 worker cores while RAMCloud requires 17.6 active cores, that is 3× more resources. For the same scenario, we also show Figure 9b that shows the aggregate active dispatch cores. Interestingly, gains are higher for dispatch, e.g., to achieve 450 KOp/s, Tailwind needs only 1/4 of dispatch cores used by RAMCloud.

Both observations confirm that, for updates, most of the resources are spent in processing replication requests. To get a better view on the impact when GET/PUT operations are mixed, we show Figure 10. It represents active worker and dispatch cores, respectively, when varying clients. When requests consist of updates only, Tailwind reduces worker cores utilization by 15% and dispatch core utilization by 50%. This is stems from the fact that a large fraction of dispatch load is due to replication requests in this case. With 50/50 reads and writes, worker utilization is slightly improved to 20% while it reaches 50% when the workload consists of 5% writes only.

Interestingly, dispatch utilization is not reduced when reducing the proportion of writes. With 5% writes Tailwind utilizes even more dispatch than RAMCloud. This is actually a good sign, since read workloads are dispatch-bound. Therefore, Tailwind allows RAMCloud to process even more reads by accelerating write operations. This is implicitly shown in Figure 10 with "Replication" graphs that represent worker utilization due to waiting for replication requests. For update-only workloads, RAMCloud spends 80% of the worker cycles in replication. With 5% writes RAMCloud spends 62% of worker cycles waiting for replication requests to complete against 49% with Tailwind. The worker load difference is spent on servicing read requests.

4.5 Scaling with Available Resources

We also investigated how Tailwind improves internal server parallelism (i.e. more cores). Figure 11 shows throughput and worker utilization with respect to available worker cores. Clients (fixed to 30) issue 50/50 reads and writes to 4 servers. Note that we do not count the dispatch core with available cores, as it is always available. With only a single worker core per machine, RAMCloud can serve 430 KOp/s compared to 660 KOp/s for Tailwind with respectively 4.5 and 3.5 worker cores utilization. RAMCloud can over-allocate resources to avoid deadlocks, which explains why it can go above the limit of available cores. Interestingly, when increasing the available worker cores, Tailwind enables better scaling. RAMCloud does not achieve more throughput with more than 5 available cores. Tailwind continues to improve throughput up to all 7 available cores per machine.

Even though both RAMCloud and Tailwind exhibit a plateau, this is actually due to the dispatch thread limit that cannot take more requests in. This suggests that Tailwind allows RAMCloud to better take advantage of per-machine parallelism. In fact, by eliminating the replication requests from dispatch, Tailwind allows more client-

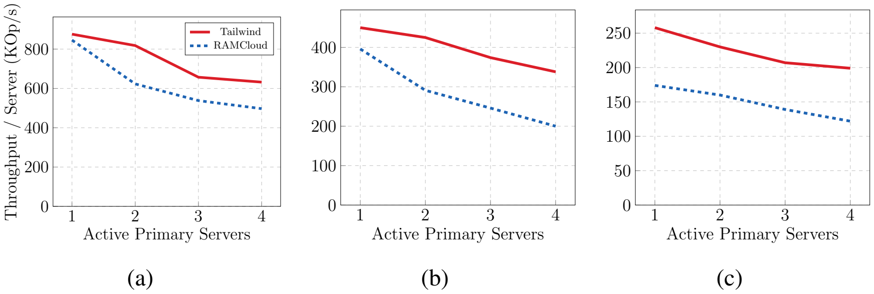


Figure 8: Throughput per active primary servers when running (a) YCSB-B (b) YCSB-A (c) WRITE-ONLY with 30 clients.

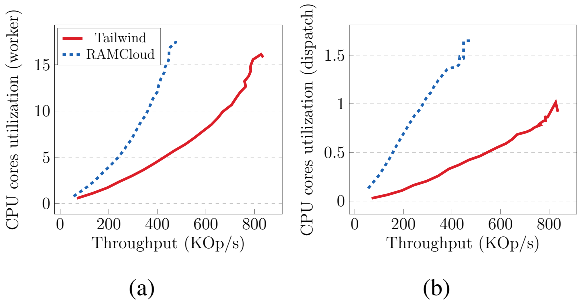


Figure 9: Total (a) worker and (b) dispatch CPU core utilization on a 4-server cluster. Clients use the WRITE-ONLY workload.

issued requests in the system.

4.6 Impact on Crash Recovery

Tailwind aims to accelerate replication while keeping strong consistency guarantees and without impacting recovery performance. Figure 12 shows Tailwind's recovery performance against RAMCloud. In this setup data is inserted into a primary replica with possibility to replicate to 10 other backups. RAMCloud's random backup selection makes it so that all backups will end up with approximately equal share of backup data. After inserting all data, the primary kills itself, triggering crash recovery.

As expected, Tailwind almost introduces no overhead. For instance, to recover 1 million 100 B objects, it takes half a second for Tailwind and 0.48 s for RAMCloud. To recover 10 million 100 B objects, Both Tailwind and RAMCloud take roughly 2.5 s.

Tailwind must reconstruct metadata during recovery (§3.4.1), but this only accounts for a small fraction of the total work of recovery. Moreover, reconstructing metadata is only necessary for open buffers, i.e. still in memory. This can be orders of magnitude faster than loading a buffer previously flushed on SSD, for example.

5 Discussion

5.1 Metadata Space Overhead

In its current implementation, Tailwind appends metadata after every write to guarantee RDMA writes atomicity (§3.1). Although this approach appears to introduce space overhead, RAMCloud's log-cleaning mechanism efficiently removes old checksums without performance impact [26]. In general, Tailwind adds only 4 bytes per object which is much smaller than, for example, RAMCloud headers (30 bytes).

5.2 Applicability

Tailwind can be used in many systems that leverage distributed logging [4, 12, 18, 20, 22, 33] provided they have access to RDMA-based networks. Recently, RDMA is supported in Ethernet in the form of RoCE or iWARP [8] and is becoming prevalent in datacenters [17, 39]. To be properly integrated in any system, Tailwind needs: (1) appending a checksum after each write; (2) implementing algorithm 1 during recovery. Aspects such as memory/buffer management do not impact Tailwind's core design nor performance gains because Tailwind reclaims replication-processing CPU cycles at backups.

6 Related Work

One-sided RDMA-based Systems. There is a wide range of systems recently introduced that leverage RDMA [4, 5, 10, 15, 16, 27, 30, 33, 34, 36, 38]. For instance, many of them use RDMA for normal-case operations. Pilaf [15] implements client-lookup operations with one-sided RDMA reads. In contrast, with HERD [10] clients use one-sided RDMA writes to send GET and PUT requests to servers, that poll their receive RDMA buffers to process requests. In RFP [10] clients use RDMA writes to send requests, and RDMA reads to poll (remotely) replies. Crail [29] uses one-sided RDMA to transfer I/O blocks, but it is not designed for availability or fault-tolerance. LITE [32] is a kernel module providing efficient one-sided operations and could be used to implement Tailwind.

Many systems also use one-sided RDMA for replication. For instance, FaRM [4, 5], HydraDB [33], and DARE [22] use one-sided RDMA writes to build a message-passing interface. Replication uses this interface to place messages in remote ring buffers. Servers have to poll these ring buffers in order to fetch messages, process them, and apply changes. In [6] authors use one-sided RDMA for VM migration. The sender asynchronously replicate data and notifies the receiver at

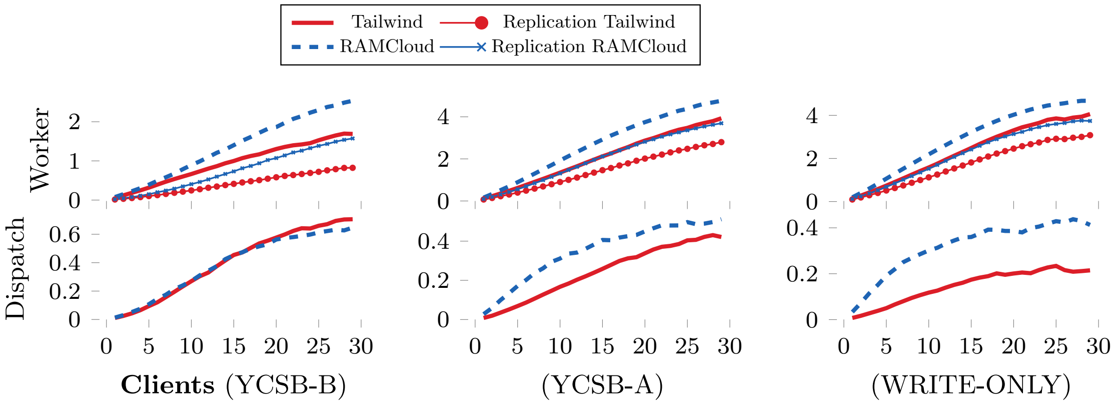


Figure 10: Total dispatch and worker cores utilization per server in a 4-server cluster. "Replication" in worker graphs represent the fraction of worker load spent on processing replication requests on primary servers.

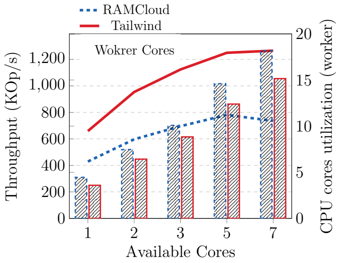


Figure 11: Throughput (lines) and total worker cores (bars) as a function of available cores per machine. Values are aggregated over 4 servers.

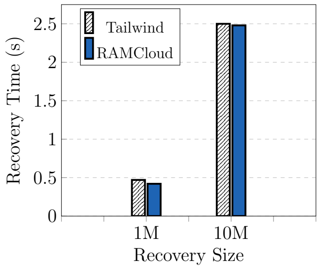


Figure 12: Time to recover 1 and 10 million objects with 10 backups.

the end of transfer then the backup applies changes in a transactional way.

No system that uses RDMA writes for replication leaves the receiver CPU completely idle. Instead, the receiver must poll receive buffers and process requests, which defeats one-sided RDMA efficiency purposes. Tailwind frees the receiver from processing requests by directly placing data to its final storage location.

Reducing Replication Overheads. Many systems try to reduce replication overheads either by relaxing/tuning consistency guarantees [3, 7, 14] or using different approaches for fault-tolerance [37]. Mojim [38] is a replication framework intended for NVMM systems. For each server it considers a mirror (backup) machine to which it will replicate all data through (two-sided) RDMA. It supports multiple levels of consistency and durability. RedBlue [14] and Correctables [7] provide different consistency levels to the applications and allows them to trade consistency for performance. Tailwind does not sacrifice consistency to improve normal-case system performance.

7 Conclusion

Tailwind is the first replication protocol that fully exploits one-sided RDMA; it improves performance without sacrificing durability, availability, or consistency. Tailwind leaves backups unaware of RDMA writes as they happen, but it provides them with a protocol to rebuild metadata in case of failures. When implemented in RAMCloud, Tailwind substantially improves throughput and latency with only a small fraction of resources originally needed by RAMCloud.

Acknowledgments

This work has been supported by the BigStorage project, funded by the European Union under the Marie Sklodowska-Curie Actions (H2020-MSCA-ITN-2014-642963), by the Spanish Ministry of Science and Innovation (contract TIN2015- 65316), by Generalitat de Catalunya (contract 2014-SGR-1051). This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1566175 and CNS-1750558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work was supported in part by Facebook and VMware.

References

[1] The OpenCompute Project. http://www.opencompute.org/.

[2] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with

ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC '10, pp. 143–154.

[3] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[4] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 401–414.

[5] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 54–70.

[6] Gerofi, B., and Ishikawa, Y. Rdma based replication of multiprocessor virtual machines over high-performance interconnects. In *2011 IEEE International Conference on Cluster Computing* (Sept 2011), pp. 35–44.

[7] Guerraoui, R., Pavlovic, M., and Seredinschi, D.-A. Incremental consistency guarantees for replicated objects. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 169–184.

[8] InfiniBand Trade Association. *IB Specification Vol 1*, 03 2015. Release-1.3.

[9] Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., and Ailamaki, A. Scalability of write-ahead logging on multicore and multisocket hardware. *The VLDB Journal 21*, 2 (Apr 2012), 239–263.

[10] Kalia, A., Kaminsky, M., and Andersen, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 295–306.

[11] Kalia, A., Kaminsky, M., and Andersen, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 185–201.

[12] Kreps, J., Narkhede, N., and Rao, J. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece* (2011).

[13] Li, B., Ruan, Z., Xiao, W., Lu, Y., Xiong, Y., Putnam, A., Chen, E., and Zhang, L. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 137–152.

[14] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 265–278.

[15] Mitchell, C., Geng, Y., and Li, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2013), USENIX ATC'13, USENIX Association, pp. 103–114.

[16] Mitchell, C., Montgomery, K., Nelson, L., Sen, S., and Li, J. Balancing CPU and network in the cell distributed b-tree store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 451–464.

[17] Mittal, R., Lam, V. T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 537–550.

[18] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17*, 1 (Mar. 1992), 94–162.

[19] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398.

[20] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., and Rosenblum, M. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 29–41.

[21] Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S., Stutsman, R., and Yang, S. The ramcloud storage system. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[22] Poke, M., and Hoefler, T. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2015), HPDC '15, ACM, pp. 107–118.

[23] Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Analysis and evolution of journaling file systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 8–8.

[24] Robert, R., and Eric, E. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *;login: 39*, 6 (2014), 36–38.

[25] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 26–52.

[26] Rumble, S. M., Kejriwal, A., and Ousterhout, J. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2014), FAST'14, USENIX Association, pp. 1–16.

[27] Shi, J., Yao, Y., Chen, R., Chen, H., and Li, F. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 317–332.

[28] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–10.

[29] Stuedi, P., Trivedi, A., Pfefferle, J., Stoica, R., Metzler, B., Ioannou, N., and Koltsidas, I. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Eng. Bull. 40* (2017), 38–49.

[30] Su, M., Zhang, M., Chen, K., Guo, Z., and Wu, Y. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 1–15.

[31] Taleb, Y., Ibrahim, S., Antoniu, G., and Cortes, T. Characterizing performance and energy-efficiency of the ramcloud storage system. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (June 2017), pp. 1488–1498.

[32] Tsai, S.-Y., and Zhang, Y. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 306–324.

[33] Wang, Y., Zhang, L., Tan, J., Li, M., Gao, Y., Guerin, X., Meng, X., and Meng, S. Hydradb: A resilient rdma-driven key-value middleware for in-memory cluster computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 22:1–22:11.

[34] Wei, X., Shi, J., Chen, Y., Chen, R., and Chen, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.

[35] Ylönen, T. Concurrent shadow paging: A new direction for database research.

[36] Yu, C., Xu, C.-Z., Liao, X., Jin, H., and Liu, H. Live virtual machine migration via asynchronous replication and state synchronization. *IEEE Transactions on Parallel Distributed Systems 22* (2011), 1986–1999.

[37] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 15–28.

[38] Zhang, Y., Yang, J., Memaripour, A., and Swanson, S. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 3–18.

[39] Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M. H., and Zhang, M. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 523–536.

On Fault Tolerance, Locality, and Optimality in Locally Repairable Codes

Oleg Kolosov[†], Gala Yadgar[*†], Matan Liram[*], Itzhak Tamo[†], and Alexander Barg[§]
[†]*School of Electrical Engineering, Tel Aviv University*
[*]*Computer Science Department, Technion*
[§]*Department of ECE/ISR, University of Maryland*

Abstract

Erasure codes are used in large-scale storage systems to allow recovery of data from a failed node. A recently developed class of erasure codes, termed *locally repairable codes (LRCs)*, offers tradeoffs between storage overhead and repair cost. LRCs facilitate more efficient recovery scenarios by storing additional parity blocks in the system, but these additional blocks may eventually increase the number of blocks that must be reconstructed. Existing codes differ in their use of the additional parity blocks, but also in their locality semantics and in the parameters for which they are defined. As a result, existing theoretical models cannot be used to directly compare different LRCs to determine which code will offer the best recovery performance, and at what cost.

In this study, we perform the first systematic comparison of existing LRC approaches. We analyze Xorbas, Azure's LRCs, and the recently proposed Optimal-LRCs in light of two new metrics: the *average degraded read cost*, and the *normalized repair cost*. We show the tradeoff between these costs and the code's fault tolerance, and that different approaches offer different choices in this tradeoff. Our experimental evaluation on a Ceph cluster deployed on Amazon EC2 further demonstrates the different effects of realistic network and storage bottlenecks on the benefit from each examined LRC approach. Despite these differences, the normalized repair cost metric can reliably identify the LRC approach that would achieve the lowest repair cost in each setup.

1 Introduction

In large-scale storage systems consisting of hundreds of thousands of servers, node failures are the norm rather than the exception. For this reason, redundancy is added to ensure availability of the data despite the failures. Typically, the redundancy of hot data is achieved by *replication* of each data object, ensuring the availability of the data as long as one replica is available. This also allows efficient reconstruction of data that was stored on a failed node from the surviving replicas.

Due to the high overhead of replication, most of the data is stored redundantly by erasure coding. With an (n,k) *erasure code*, the data is split into k *data blocks* that are used to generate $n-k$ *parity blocks*. The blocks are distributed across n different nodes, so that the original data can be reconstructed as long as at least k blocks are available. The *storage overhead* of erasure coding is $\frac{n}{k}$ —considerably lower than that of replication. However, reconstruction of one data block requires reading k surviving blocks—an overhead considerably higher than that of replication.

Storage systems distinguish between two types of node failures. *Transient failures* may be caused by system restarts or updates, after which the node is available again. During this time, read operations of data stored on the failed node are served as *degraded reads*—only the required data blocks are reconstructed from the surviving blocks. *Permanent failures* occur when the node malfunctions and is no longer accessible. Typically, a failure is considered permanent after 15 minutes of unavailability, which triggers full recovery of its data. Recent studies indicate that transient failures comprise 90% of failure events [28], and only the remaining 10% trigger full node recovery. Nevertheless, recovery traffic incurs significant load on the data center's servers and network—up to 180TB of data transfer between racks each day, according to a recent study on Facebook's data centers [24].

The vast majority of failures (up to 98% [24]) constitute exactly one unavailable node. Thus, several approaches have been used to design erasure codes that can withstand several concurrent failures, but optimize the recovery cost of a single node. These include preprocessing the surviving data to minimize repair network bandwidth [6,22,25], and reducing the amount of data read from each surviving node [7, 9, 12, 14, 23, 38]. These codes can reduce the amount of data read by up to 50%, but in many realistic settings, this reduction is no more than 25%, or not applicable due to the required I/O granularity [18].

A different approach increases the storage overhead and utilizes the added redundancy to optimize the recovery of a single data node. An (n,k,r) *locally repairable code (LRC)* supports the *local* recovery of an unavailable block by reading at most r surviving blocks. These codes were originally designed to reduce the cost of degraded reads, and thus most of them optimize only the recovery of data blocks [10,11]. Others further optimize the recov-

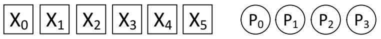


Figure 1: (10,6) Reed-Solomon

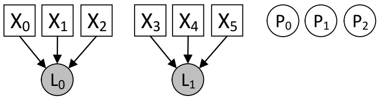


Figure 2: (11,6,3) Azure-LRC

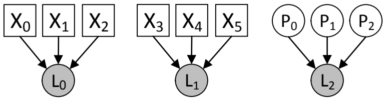


Figure 3: (12,6,3) Optimal-LRC

ery of all of the parity blocks, but do so for a limited set of system parameters [28]. In a recent work [30], new codes were constructed that support local recovery of both data and parity blocks, with the same storage overhead as previously known constructions.

LRCs present an inherent tradeoff. On the one hand, they considerably reduce the amount of data that must be read for degraded reads and recovery. On the other hand, in order to store the additional parity, the system must store more blocks on each of its nodes, or allocate more nodes for the same amount of data. In the first case, more data must be reconstructed whenever a node fails, while the latter increases the probability of failure in the system. As a result, LRCs not only increase the system's storage overhead, but might also increase its overall recovery costs. Different codes offer different tradeoffs between storage overhead and recovery cost, and between recovery cost and the cost of degraded reads. Furthermore, they are defined for different (n,k,r) combinations and differ in their locality semantics. Thus, directly comparing their costs and benefits is a nontrivial task, which makes it hard to choose the optimal code and configuration for a given system.

In this study, we perform the first comprehensive analysis of the different LRC approaches. We take into account the overall cost of recovery, including data and parity blocks, as well as the maximum number of failed blocks the code can recover. Our analysis includes *Xorbas* [28], *Azure-LRC*—the LRC codes used by Microsoft Azure [11], and *Optimal-LRC*—a recently proposed theoretically optimal code [30]. We also define a new code, *Azure-LRC+1*, which is based on Azure-LRC and supports efficient recovery of all parity blocks.

We conduct a theoretical comparison between the different LRC approaches. Our analysis demonstrates the limitations of existing measures, such as locality and average repair cost. Thus, we define new metrics that model each code's overhead, full-node repair cost, degraded read cost, and fault tolerance. Our results demonstrate the tradeoff between the objectives measured by these costs, and how different codes optimize different objectives.

We follow the theoretical analysis with an evaluation of these codes in a Ceph cluster deployed in AWS EC2. Our experimental evaluation shows that we can accurately predict the amount of data required by each code for reconstructing an entire storage node. This prediction also provides a good estimate of the time required for reconstruction, for most combinations of storage type, network configurations, and foreground traffic.

The rest of this paper is organized as follows. Section 2 presents LRCs and motivates our analysis. We describe our new LRC and metrics in Section 3, and our theoretical analysis in Section 4. Our system-level setup is described in Section 5, with the evaluation in Section 6. We survery related work in Section 7, and conclude in Section 8.

2 Preliminaries

The storage overhead of an erasure code is defined as $\frac{n}{k}$. Its *minimal distance*, d, is defined as the smallest number of concurrent node failures that may cause data loss. In other words, there is at least one combination of d node failures from which the code will not be able to recover the data. An important class of codes, termed *maximum distance separable (MDS)* codes, is characterized by the relation $d = n-k+1$, and provides the largest possible d for given n and k. In MDS codes, k surviving blocks are required to recover a failed block. Reed-Solomon codes [26] are the most commonly used MDS codes owing to their parameter flexibility and efficient implementation.

An (n,k,r) locally repairable code (LRC) consists of k data blocks and $n-k$ parity blocks. The data blocks are grouped into *local groups* no larger than r. A *local parity* is computed from each local group of blocks and can be used for the recovery of any block in this group. In total, each local group of LRC contains at most $r+1$ blocks. In case of an arbitrary failure of one block in a local group, r surviving blocks are required for its recovery. A *global parity* is a function of all data blocks, and can thus be used to recover any lost block. Pyramid codes [10], which are based on (n,k) Reed-Solomon codes were the first suggested family of LRCs. Another family, Azure-LRC, is a variation of Pyramid codes and is used in Windows Azure [11].

Figure 1 depicts a (10,6) Reed-Solomon code, and Figure 2 shows the (11,6,3) Azure-LRC which results from replacing one of its global parities with two local parities. In this example, P_3 was replaced with L_0 and L_1, which can be used in the recovery of groups (X_0,X_1,X_2) and (X_3,X_4,X_5), respectively. In the new code, any of the data blocks can be repaired by reading the remaining three blocks in its local group. Thus, the recovery cost of a data block is reduced by 50%. However, the overhead increases by 10%, from $\frac{10}{6}$ to $\frac{11}{6}$. Note also that the new code is non-MDS: it can repair any four missing blocks but not any five, therefor $d = 5$, but $n-k+1 = 6$.

Azure-LRC successfully reduces the repair cost of data blocks and local parities, and, as a result, the degraded

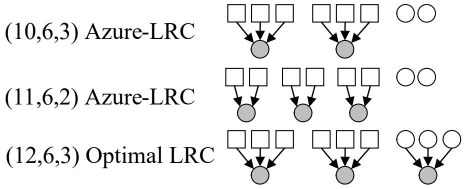


Figure 4: Three LRCs with the same k, demonstrating different tradeoffs between locality and overhead.

read cost. However, due to the allocation of blocks to nodes, when an entire node must be reconstructed, this node will include a significant number of global parities, which will require k surviving blocks for recovery. For example, in (11,6,3) Azure-LRC, which contains $m = 3$ global parities, an average of $\frac{3}{11} = 27.2\%$ of the blocks stored on each node will be global parities.

Coding theory distinguishes between two types of (n,k,r) LRCs. In codes with *information-symbol locality*, only the data blocks can be repaired in a local fashion by r surviving blocks, while the global parities require k blocks for recovery. We refer to these codes as *data-LRCs*. Pyramid and Azure-LRC are data-LRCs. In contrast, in codes with *all-symbol locality*, all the blocks, including the global parities, can be repaired locally from r surviving blocks. We refer to such codes as *full-LRCs*.

Optimal-LRC is a recently proposed full-LRC [30]. In this code, k data blocks and m global parities are divided into groups of size r, and a local parity is added to each group, allowing repair of *any* lost block by the r surviving blocks in its group. r does not necessarily divide $m+k$, but Optimal-LRC requires that $n \bmod (r+1) \neq 1$. Figure 3 shows a (12,6,3) Optimal-LRC. Each of the global parities, P_0, P_1, and P_2, can be reconstructed from the other global parities and the local parity L_2. The overhead of this code is higher than that of the (11,6,3) Azure-LRC in Figure 2, but its minimum distance is also higher ($d = 6$).

Full-LRCs introduce a new point in the tradeoff between fault tolerance and performance, which previously consisted only of MDS codes and data-LRCs. Gopalan et al. [8] proved that an upper bound on the minimal distance for an (n,k,r) LRC is $d \leq n - k - \lceil \frac{k}{r} \rceil + 2$. Codes that achieve this bound are regarded as optimal; in particular, Optimal-LRC has been shown to achieve this bound. Specifically, the minimum distance of Optimal-LRC was shown to be [30]:

$$d = n - k - \left\lceil \frac{k}{r} \right\rceil + 2, \quad \text{if } (r+1)|n$$

$$d \geq n - k - \left\lceil \frac{k}{r} \right\rceil + 1, \quad \text{if } (r+1) \not| n, \quad r|(k+1).$$

Challenges. Azure-LRC provides an appealing tradeoff when compared to Reed-Solomon codes: a 10% increase in storage overhead can halve the cost of all degraded reads and most block repairs. Unfortunately, the comparison between data-LRCs and full-LRCs is not similarly straightforward. Consider, for example, the three codes in Figure 4. The (11,6,2) Azure-LRC has three local parities, one more than the (10,6,3) Azure-LRC, which reduces its r from 3 to 2, but increases its overhead by 10%. The (12,6,3) Optimal-LRC also has three local parities. However, rather than reducing r, the additional local parity enables local repair of the global parities. Thus, r represents different locality semantics in each of these models. In addition, each model represents a different tradeoff between the cost of degraded read and the cost of full node repair, and between these costs and the overhead.

It is not entirely clear which of these codes will have the lowest repair cost. Clearly, r alone cannot serve as a metric for comparing data-LRCs to full-LRCs. The *average repair cost (ARC)* used in previous analyses [11] fails to capture the effect the code's overhead has on its repair cost. In the next section, we introduce three composite metrics that facilitate a systematic comparison of LRCs.

The task of comparing different codes is further complicated by the fact that existing codes are not all defined for the same range of parameters. Our new metrics alleviate this problem to some extent. To eliminate the problem completely, we adopt a somewhat 'flexible' interpretation of Azure-LRC. We also use a new construction of Optimal-LRC, which is optimal for parameters for which an explicit construction has not been given before.

Finally, theoretically proven benefits are not always achievable in real systems. The repair-cost benefit of different codes may be determined by factors such as storage and network bandwidth, the nature and priority of the foreground load, and the system-level implementation. Thus, we complement our theoretical analysis with an evaluation on a distributed cluster in Amazon EC2, where we verify our metrics and identify additional factors that should be taken into account when designing an erasure coded storage system.

3 Methodology

Metrics. The starting point of our theoretical analysis consists of the existing measures described above: r is the maximal number of blocks required for the recovery of any block or a data block, in full-LRCs and data-LRCs, respectively. The *overhead* of the code is $\frac{n}{k}$, and its minimal distance is d. We use d to represent the code's fault tolerance, despite its inherent limitation—two codes with the same d may be considered equally fault tolerant, although one may prevent data loss in more combinations of correlated failures than the other [11]. The *mean time to data loss (MTTDL)* is considered a more accurate measure for fault tolerance. However, to calculate the MTTDL of a code, one must construct a Markov chain for every specific set of n,k,r parameters. In addition, this model does not always yield an analytic closed-form equation. Thus, d is more appropriate for our large-scale analysis. For a limited comparison of a small set of constructions, d can be replaced with MTTDL.

The average repair cost (ARC) has been used in previous studies [11], and is based on the assumption that the probability of repair due to failure is the same for all blocks. It is defined as

$$ARC = \frac{\sum_{i=1}^{n} cost(b_i)}{n},$$

where b_i is the ith block in the code, and $cost(b_i)$ is the number of blocks required for the repair of b_i. For example, the ARC of the (10,6,3) Azure-LRC in Figure 4 is $\frac{(8\times3)+(2\times6)}{10} = 3.6$. Similarly, in the same figure, the ARC of the (11,6,2) Azure-LRC is 2.73 and that of the (12,6,3) Optimal-LRC is 3.

ARC does not take into account the higher overhead of some of these codes, which implies that more blocks will have to be repaired in the event of a node failure. We address this by defining a new composite metric for the cost of full-node repair. The *normalized repair cost (NRC)* of a code is the product of its ARC and overhead:

$$NRC = ARC \times \frac{n}{k} = \frac{\sum_{i=1}^{n} cost(b_i)}{k}.$$

NRC can also be viewed as the average cost of repairing a failed data block, where the cost of repairing the parity blocks is amortized over the k data blocks. For example, the NRC of the (10,6,3) Azure-LRC in Figure 4 is $\frac{(8\times3)+(2\times6)}{6} = 6$. Similarly, the NRC of the (11,6,2) Azure-LRC is 5 and that of the (12,6,3) Optimal-LRC is 6.

ARC is also inappropriate for modeling the cost of degraded reads. By definition, degraded reads refer to data blocks only, while ARC averages the repair cost of all blocks—data and parity alike. We define the *average degraded read cost* (*'degraded cost'*, in short) as the average cost of repairing data blocks only:

$$Degraded\ cost = \frac{\sum_{i=1}^{k} cost(b_i)}{k},$$

where blocks $b_1,...,b_k$ are the object's data blocks. For example, the degraded cost of the (10,6,3) Azure-LRC and the (12,6,3) Optimal-LRC in Figure 4 is $\frac{6\times3}{6} = 3$. Similarly, the degraded cost of the (11,6,2) Azure-LRC is 2. Note that in the general case, the degraded cost is not always equal to r.

We base our analysis on three existing LRCs: Xorbas, Azure-LRC, and Optimal-LRC. We use Reed-Solomon codes as a baseline for some of our comparisons. Below, we describe how we extended the definitions of these codes for our analysis and evaluation.

Xorbas. Xorbas [28] is a full-LRC, in which the global parities can be recovered from the local parities. Figure 5 shows a (16,10,5) Xorbas code. Each local parity belongs to a group containing five data blocks. The special construction of Xorbas ensures that any of the global parities can be reconstructed by the remaining global parities and the two local parities. Thus, $r = 5$ for *all* the blocks in the code. This special property can be maintained if we remove the same number of blocks from each group. For example, a (13,8,4) Xorbas code can be obtained by removing two data blocks and one global parity from the original construction. The number of global parities can be further reduced to achieve a lower overhead, without reducing r. For example, a (12,8,4) Xorbas code has the same r as the (13,8,4) code, but a smaller d.

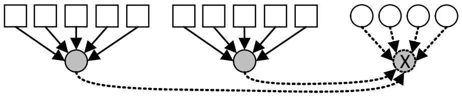


Figure 5: (16,10,5) Xorbas. ⊗ marks a function computed by the local parities, not a real block.

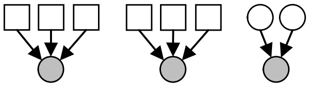


Figure 6: (11,6,3) Azure-LRC+1

Azure-LRC. We use Azure-LRC as the data-LRC in our evaluation. It is explicitly defined in its original paper only for (n,k,r) where r divides k, and the number of local parities is $l = \frac{k}{r}$ [11]. For the sake of analysis, we extend this code to the general case as follows. In an (n,k,r) Azure-LRC where r does not divide k, the number of local parities is $l = \lceil \frac{k}{r} \rceil$. $l-1$ groups contain r data blocks and one local parity. The remaining group contains $k \mod r$ data blocks and one local parity. For the code to have at least one global parity, we must only ensure that $k+l < n$. Although this extension results in asymmetric allocation of data blocks to groups, it allows us to consider Azure-LRC in most (n,k,r) combinations.

Azure-LRC+1. For the sake of analysis, we define a new full-LRC which is based on Azure-LRC. An (n,k,r) *Azure-LRC+1* code is constructed by adding one local parity to the group of global parities of an $(n-1,k,r)$ Azure-LRC. This local parity is computed as the XOR of all the global parities. Figure 6 shows an (11,6,3) Azure-LRC+1 constructed from the (10,6,3) Azure-LRC in Figure 4. When one global parity block is missing, it can be repaired from the remaining global parities and the additional local parity. Thus, Azure-LRC+1 will have $l+1$ local parities, $l = \lceil \frac{k}{r} \rceil$, and can be constructed as long as $k+l+1 < n$. This naïve definition implies that an Azure-LRC+1 construction may result in a local parity added to a 'group' of one global parity. Nevertheless, it has the added value of being directly and easily applicable to any system that uses Azure-LRC or Pyramid codes, and is thus an important aspect of our analysis.

Optimal-LRC. The original Optimal-LRC construction [30] was shown to be optimal for the cases described in Section 2. However, extending this construction to all admissible (n,k,r) combinations results in a code with lower d, which is suboptimal. To address this issue, we devised a new construction of codes in the spirit of the original construction. The advantage of the new construction is that it applies to all parameters n,k,r such that $n \bmod (r+1) \neq 1$. Furthermore, it can be shown that our new construction attains the largest possible minimum distance even when the upper bound $n-k-\lceil\frac{k}{r}\rceil+2$ is not attainable. In summary, the new construction is the first optimal construction for *all* admissible parameters. The construction and the proof of its optimality can be found in [13].

Evaluation parameters. We computed the ARC, NRC, degraded cost, overhead, and d for each of the codes described above, for all (n,k,r) combinations for which they are defined, where $9 \leq n \leq 19$ and $\frac{n}{k} \leq 2$. These combinations include specific sets of parameters that appear in the literature and in documented deployments: (18,12,3) Azure-LRC [11], (16,10,5) Xorbas, (14,10) Reed-Solomon [24], and (9,6) Reed-Solomon [37]. Due to space constraints, and for clarity of presentation, we show only results for $12 \leq n \leq 18$ and $\frac{n}{k} \leq 1.6$, which include the more common combinations. This range of parameters suffices for demonstrating our observations, which we verified on the complete range.

4 Theoretical Analysis

Figure 7 shows NRC and the degraded read cost of the different codes. For the same n, k, and r, the degraded cost is usually the same for all codes. It is different when r does not divide k, where the codes differ in their allocation of blocks to groups. As we expected, for the same n and k, increasing r increases each code's degraded read cost and NRC. However, when comparing different codes, neither r nor the degraded read cost can indicate which code will have the lowest full-node repair cost. For example, the NRC of (14,10,6) Azure-LRC+1 is lower than that of (14,10,5) Azure-LRC, although its degraded cost is higher.

Figure 8 shows the minimum distance, d, of the different codes. Figures 7 and 8 together demonstrate a clear tradeoff between repair costs and fault tolerance. In general, for given n, k, and r, to increase d one must either increase n or increase r, thus increasing both the degraded cost and NRC. Nevertheless, different codes offer different points in this tradeoff.

Data-LRC vs. full-LRC. For the same (n,k,r), there is always one full-LRC with a lower NRC than that of Azure-LRC. However, in most settings, the reduction in NRC is coupled with a reduction in d. In the settings in which it is defined, Xorbas achieves the same d but a higher NRC than Azure-LRC+1 and Optimal-LRC. Optimal-LRC and Azure-LRC+1 achieve the same d and NRC in many settings. In the settings where the NRC of Azure-LRC+1 is lower than that of Optimal-LRC, its d is also lower (except for a few corner cases discussed below).

In Figure 9, we compare the NRC of (n,k,r) Azure-LRC to that of the $(n+1,k,r)$ full-LRCs with the same d. The full-LRCs use an additional local parity to allow fast repair of the global parities. This addition always reduces the repair cost, despite the increase in storage overhead.

Optimality of Optimal-LRC. Despite its optimal properties, our analysis reveals that for a given (n,k,r), Optimal-LRC does not always achieve the lowest NRC. Optimal-LRC is designed to accommodate the global parities with the data blocks in the same group. However, when the number of global parities is much smaller than r, this results in increasing the size of one of the groups, thus increasing the NRC. For example, Figure 10 shows a (12,8,5) Azure-LRC whose NRC is lower than that of (12,8,5) Optimal-LRC, and a (16,10,6) Azure-LRC+1 whose NRC is lower than that of (16,10,6) Optimal-LRC. In both cases, Optimal-LRC can achieve a lower NRC with a smaller r, possibly at the cost of reducing d.

NRC vs. d. Our results demonstrate a subtle tradeoff between repair cost (NRC) and d. Codes with the same (n,k,r) may or may not have the same d, and are thus not directly comparable: one may satisfy fault tolerance requirements that the other does not. To facilitate a more systematic comparison, we defined another composite metric, *repair-distance ratio (rd-ratio)*, $\frac{NRC}{d}$. This can be viewed as a measure of the efficiency with which a code allocates its local parities, with the conflicting objectives of maximizing d and minimizing *NRC*.

Figure 11 shows the *rd*-ratio of all LRCs. It shows that the code with the lowest *rd*-ratio is different for different (n,k,r) combinations, and is not necessarily a full-LRC. For example, when (n,k,r) is (14,10,5), Azure-LRC has the lowest *rd*-ratio. Another interesting observation is that when fixing n and k, different codes achieve their minimal *rd*-ratio with different values of r. For example, the *rd*-ratio of (17,12,5) Optimal-LRC is lower than that of (17,12,4) Optimal LRC. When we fix (n,k) and consider the "best" r for each code, we observe that Optimal-LRC achieves the lowest *rd*-ratio. This demonstrates that this code is optimal in its allocation of local parity blocks—it efficiently reduces the repair cost with minimal reduction in d. The *rd*-ratio can be generalized to reflect different weights of NRC and d, e.g., by defining it as $\frac{NRC}{d^x}$.

Target fault tolerance. The required fault tolerance in a distributed storage system is determined by many factors, including the number of nodes, their organization into racks and clusters, and the anticipated causes of failure. Nevertheless, once the required level of fault tolerance is determined, the goal is to select a code which will provide this level at the lowest cost. In this context,

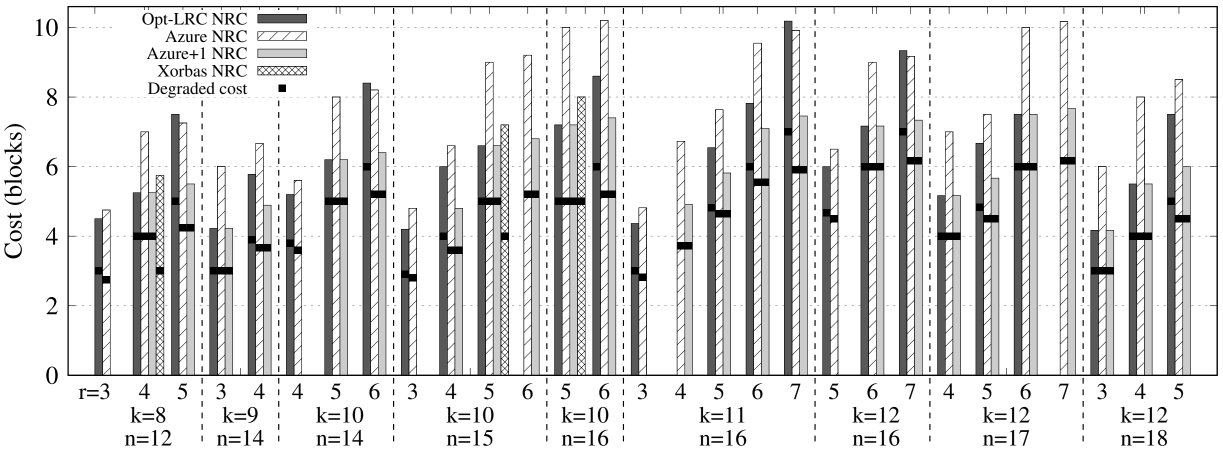


Figure 7: NRC and the degraded cost for all the codes in our evaluation. The repair cost of the full-LRCs is always lower than that of Azure-LRC.

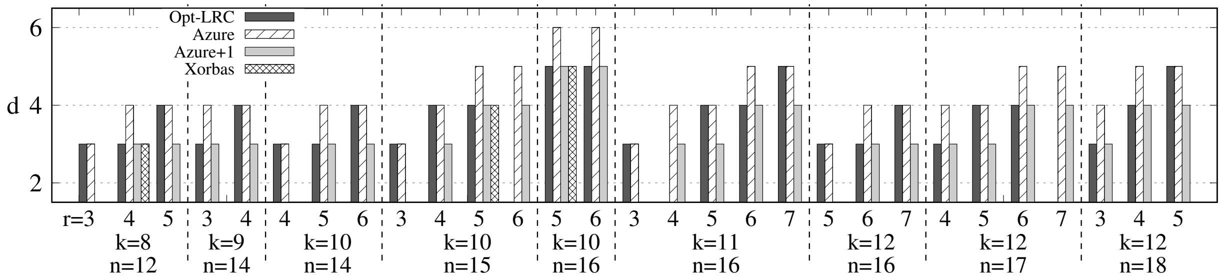


Figure 8: *d* for all the codes in our evaluation.

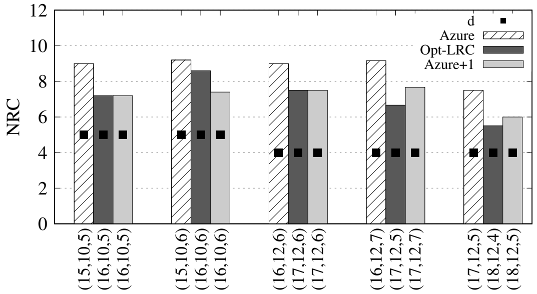


Figure 9: NRC for (n,k,r) Azure-LRC and $(n+1,k,r)$ Azure-LRC+1 and Optimal-LRC. Adding a local parity always reduces repair cost, despite the increase in overhead.

the code with the lowest *rd*-ratio may not be the optimal choice—a different code may have a higher ratio but provide the required level of fault tolerance at a lower overhead or repair cost.

We defined a threshold value of d, d_{th}, and compared the NRC of all the codes for which $d \geq d_{th}$. We considered $d_{th} \in \{3,4,5\}$, corresponding to the minimum distance of commonly deployed configurations. Figure 12 shows the NRC of all the LRCs whose $d \geq 4$. Many constructions do not provide the required fault tolerance, and are thus absent from this figure. Different codes achieved the lowest NRC for different k,n combinations. However, we note that a construction of Azure-LRC and Optimal-LRC with the required d was defined for every k,n combination. This demonstrates the flexibility of both codes. We observed similar results when setting the threshold d_{th} to 3 or 5, where increasing the threshold removed more codes from the comparison, and vice versa.

Our theoretical evaluation results demonstrate the challenges in comparing different LRC codes and approaches. Our metrics, NRC, degraded cost, and *rd*-ratio, provide a framework for directly comparing all codes in all parameter combinations. Our comparison demonstrates the benefit of full-LRCs, the flexibility of Optimal-LRC, and the realistic settings in which they may reduce the amount of data read and thus the system repair cost. In the following, we extend our notion of 'repair cost' to additional performance measures.

5 System-Level Evaluation Setup

The goal of our system-level evaluation was threefold: to validate the accuracy of NRC when predicting the amount of data read for node reconstruction, to evaluate its ability to estimate repair time and bandwidth, and to compare the recovery efficiency of the different LRCs in a real system. We omitted the minimum distance, d, from this part of our analysis, because it is not measured empirically. We focused on four representative (n,k) combinations, and compared Reed-Solomon codes, Azure-LRC, Azure-LRC+1, and Optimal-LRC in these setups. We excluded Xorbas from this part of our analysis due to design limi-

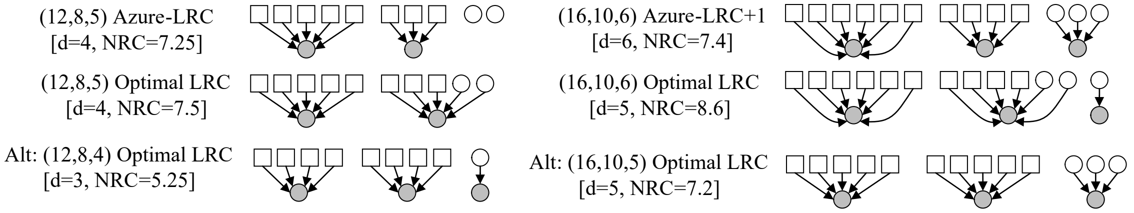


Figure 10: Examples where (n,k,r) Optimal-LRC does not achieve the lowest NRC. In both cases, an alternative $(n,k,r-1)$ Optimal-LRC achieves a lower NRC, possibly at the cost of reducing d.

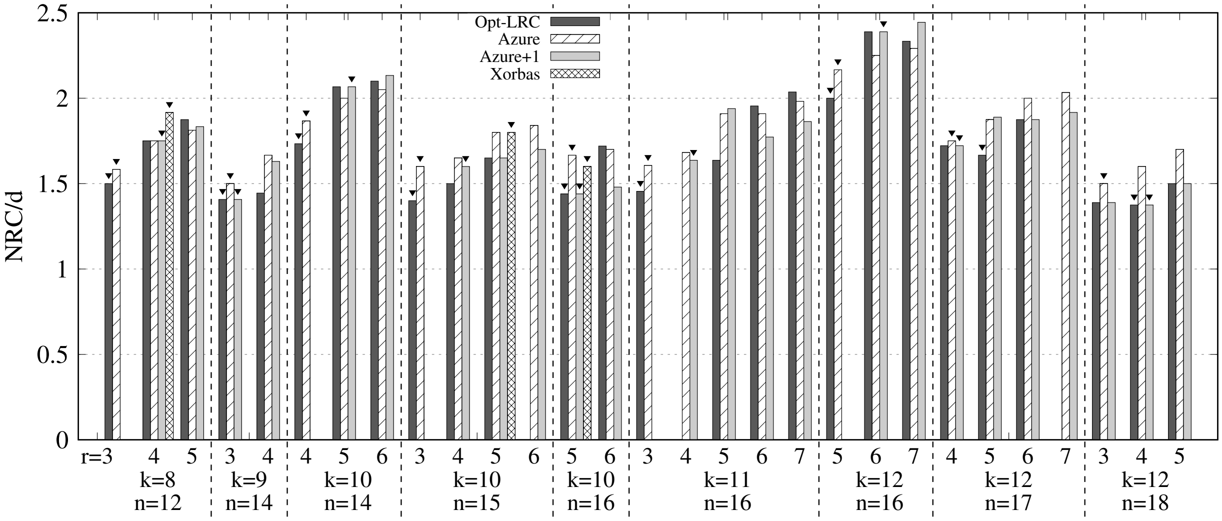


Figure 11: Repair-distance ratio ($\frac{NRC}{d}$). For each (n,k), different codes achieve their minimal rd-ratio (marked by the small triangle) with different values of r.

tations described below.

We performed our evaluation in Ceph—a distributed open-source storage system [34]. Ceph's object storage service, RADOS [36], is responsible for object placement, failure detection and failure recovery. Ceph's nodes are called *object storage devices (OSDs)*. Objects in Ceph are assigned to *placement groups*, which define the allocation of blocks to OSDs. The mapping of placement groups to OSDs is implemented by a pseudo-random mechanism, CRUSH, to ensure load balancing [35].

The *primary OSD* in each placement group is responsible for encoding the data and distributing the data and parity blocks to the remaining, *secondary*, OSDs. When one of the OSDs in a placement group fails, a *replacement OSD* is assigned to it. The primary OSD is responsible for reading the required data from the surviving OSDs, reconstructing the missing block, and sending it to the replacement OSD for permanent storage.

Ceph's design imposes certain limitations on our evaluation. When the failed OSD and the primary OSD belong to different *locality groups*, the repair data must be transferred across groups. In complex network topologies, this might incur cross-rack or cross-zone traffic that LRCs were designed to avoid. In addition, degraded reads are currently implemented by reconstructing the entire object at the primary OSD. This means that all k data blocks are read, even if only r blocks are required to repair the missing block. As a result, for degraded reads, there is no observable difference between MDS codes and LRCs.

We chose to use Ceph despite these limitations. As far as we know, it is the only open-source distributed storage system that implements LRCs as part of its main distribution. Furthermore, at the time we began this research, it was the only system to support online erasure coding, without requiring that objects are first replicated and then erasure-coded in the background.

LRC plugin. In Ceph, erasure codes are implemented as plugins. We used the Jerasure Erasure Code plugin [4], which contains an implementation of Reed-Solomon based on the Jerasure [21] and GF-Complete [20] libraries. We used the Locally Repairable Erasure Code plugin (LRC plugin) [5] to implement Azure-LRC and Azure-LRC+1[1]. This plugin first attempts to reconstruct the missing blocks from the surviving blocks in its locality group. If the block does not belong to any group, or if other blocks in the group are unavailable, it will be reconstructed from the global parities.

We made two adjustments in the LRC plugin. First,

[1] Ceph's LRC plugin actually implements Pyramid codes, and not Azure-LRC. However, the data read by these two codes in all single-node failure scenarios is identical. The precise parity calculations and fault tolerance of Azure-LRC are outside the scope of this study.

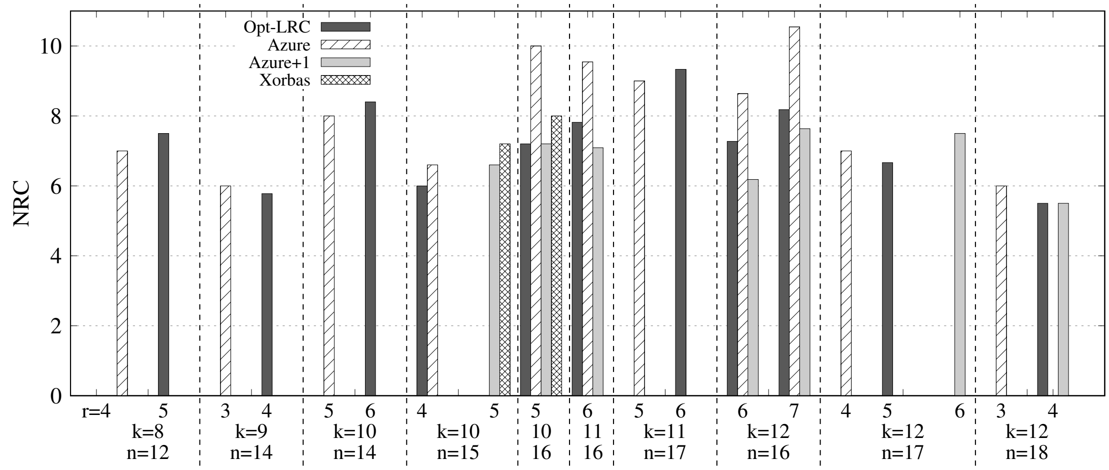


Figure 12: NRC of codes with $d \geq 4$. Azure-LRC and Optimal-LRC are the most flexible codes, defined for all (k,n) combinations.

we prevented CRUSH from rebalancing the placement groups after a node failure. This prevented the primary OSD from reading data that was not strictly required for repair. Second, we adjusted the LRC plugin to read the minimum amount of required blocks for recovery with global parities. The original implementation greedily reads *all* the remaining blocks in the stripe, which artificially increased the repair cost.

Optimal-LRC implementation. We implemented the encoding in Optimal-LRC as a multiplication by a $k \times n$ generator matrix created from the polynomial described in [13]. When creating the matrix, we transformed the k columns corresponding to the data blocks to ensure *systematic encoding* in which the data blocks are not encoded and are stored on the storage nodes in their original form. We further optimized the generator matrix to ensure that the decoding process of local recovery would consist only of XOR operations, avoiding finite field operations. We used Matlab to construct the generator matrix for each (n,k,r) Optimal-LRC in our evaluation.[2]. Beyond these initial calculations, the encoding and decoding processes of Optimal-LRC are equivalent to those of the original Ceph LRC implementation. The differences in encoding and decoding complexities are negligible compared to the I/O and network times of a large-scale storage system [6, 11, 12, 15, 19]. Similarly, there is no significant difference in the overhead of their implementation and metadata storage and maintenance.

Amazon EC2 deployment. We deployed our Ceph cluster on 20 instances in the Amazon Elastic Compute Cloud (EC2). We used `t2.medium` instances, each equipped with two Intel Xeon processors and 4GiB RAM [2]. We allocated two storage volumes to each instance, and used them to initialize two OSDs, resulting in a cluster of 40 OSDs. An additional `t2.2xlarge` instance hosted the monitor, metadata server, and client.

EC2 data centers belong to different *regions*, which correspond to distinct geographical locations. Each region contains several *availability zones*, which are connected by low latency links and guarantee failure tolerance within the region [3]. We deployed our cluster in a single availability zone in the Frankfurt region in all experiments except the multi-zone one. Unless stated otherwise, we use General Purpose SSD as storage devices [1].

In our basic "node repair" experiment, we populated the cluster with 200GB of data, written as 64MB objects. These objects are distributed across 512 placement groups. Thus, each OSD stored, on average, 5GB of data, and additional parity blocks according to the evaluated code. We killed one OSD daemon on one instance and removed this OSD from the cluster. This initialized the repair process, which was performed by the primary OSD in each affected placement group. We recorded the amount of data read from each device and the CPU utilization of each instance, until the full recovery of the cluster. We describe variations of this experiment with foreground workload and with slower storage below.

6 Results

Amount of data read and transferred. Figure 13 shows the number of blocks read by each code during repair, normalized to the number of data blocks on the failed OSD. We also present the ARC and NRC of each code, for comparison. We use an (n,k) Reed-Solomon in each configuration as our baseline. The results show the considerable reduction in repair cost achieved by LRCs, and that full-LRCs achieve a larger reduction, as shown in our theoretical evaluation.

For a given (n,k,r) combination, both ARC and NRC can predict which code will incur the the highest and lowest repair costs. At the same time, they are both inaccurate in their prediction of the actual repair cost. The reason for this inaccuracy is different for each metric. ARC in-

[2] Our implementation of Optimal-LRC in Ceph and its construction in Matlab will be made available as open-source projects.

Code	ARC	Adjusted ARC	NRC	Adjusted NRC	Data read	Time (s)
(13,10) RS	10	10	13	13	11.71	89
(14,10,5) Azure	5.71 (0.57)	5.5 (0.55)	8 (0.62)	7.7 (0.59)	6.74 (0.58)	59 (0.66)
(14,10,6) Azure	5.85 (0.58)	5.6 (0.56)	8.2 (0.63)	7.84 (0.6)	6.86 (0.59)	57 (0.64)
(15,10,5) Azure+1	4.4 (0.44)	4.5 (0.45)	6.6 (0.51)	6.75 (0.52)	5.96 (0.51)	57 (0.64)
(15,10,6) Azure+1	4.53 (0.53)	4.59 (0.46)	6.8 (0.52)	6.88 (0.53)	6.08 (0.52)	57 (0.64)

Table 1: Adjusted ARC and NRC according to block distribution on the failed OSD. The adjusted NRC corresponds to the actual amount of data read for recovery. The values in parenthesis show the costs normalized to Reed-Solomon with the same *k* and *d*.

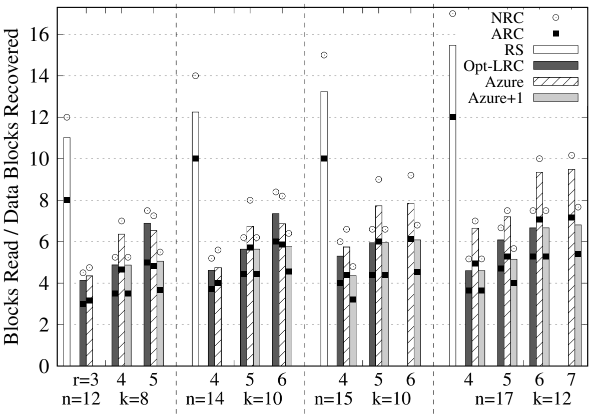


Figure 13: The number of average read blocks per data block repaired, compared to expected ARC and NRC.

herently underestimates the absolute cost, because it does not take into account the code's overhead. As a result, it is not useful for comparing codes with different storage overheads. For example, the ARC of (14,10) Reed-Solomon and (15,10) Reed-Solomon is 10 for both, but they read 12.53 and 13.24 blocks per data block recovered, respectively.

The inaccuracy of NRC is the result of our limited evaluation setup. Although CRUSH attempts to uniformly distribute data and parity blocks on all OSDs, its mapping is deterministic, and the actual distribution with 40 OSDs is not perfectly uniform. As a result, some OSDs store more blocks than others, and the percentage of data and parity blocks on each OSD is different. We verified that this is the cause of the inaccuracy - by distinguishing between the blocks on the failed OSD according to the number of blocks required for their repair and observing that their percentage is different than expected. We adjusted the NRC and ARC in several setups according to the observed distribution, although we still assumed 5GB of data on each OSD[3].

Table 1 shows the detailed metric and results for Reed-Solomon, Azure-LRC, and Azure-LRC+1, when $k = 10$, and $d = 4$. The required storage overhead is different for each code, which makes it difficult to directly compare their repair costs. The mapping of OSDs to placement groups is also different for each *n*. The table shows the calculated and the adjusted ARC and NRC of all codes,

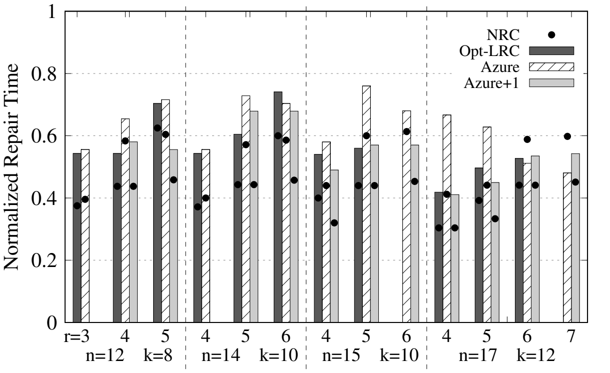


Figure 14: Recovery time of LRCs normalized to Reed-Solomon with the same *k* and *n*.

with the repair cost of each LRC compared to that of Reed-Solomon in parenthesis. The adjusted NRC provides a fairly accurate prediction of the amount of data read for recovery (we discuss the recovery time below). This confirms that in a large-scale storage system with uniform block distribution, the NRC can accurately predict the average repair cost of an entire storage node.

The amount of data transferred between nodes was almost identical to the amount of data read. We verified that the differences were caused by the role of the primary OSD in the reconstruction process: when the primary stored one of the blocks required for reconstruction, it did not have to transfer this block to another OSD. On the other hand, the primary always had to transfer the reconstructed block to the replacement OSD. In light of this simple correlation, we omit the amount of data transferred from the rest of our discussion.

Repair time. Figure 14 shows the recovery time of LRCs normalized to Reed-Solomon with the same *k* and *n* (normalizing to Reed-Solomon with the same *d* yields equivalent results). Our results show that the reduction in the amount of data read for repair does not directly translate to a reduction in repair time. This is the result of additional bottlenecks in the system, such as queuing and batching delays. We verified that the CPU utilization is the same for all codes, ruling out encoding costs as a bottleneck. However, the I/O bandwidth utilized by the codes was slightly different. Reed-Solomon typically achieved a higher throughput than the LRCs—it reads considerably more data than the other codes, which allows it to saturate the storage devices. Thus, the reduction in repair time achieved by the LRCs was smaller than that predicted by NRC. Overall, the full-LRCs achieved the greatest reduc-

[3] Ceph does not report the number of data blocks stored on each OSD, and we could not distinguish between data blocks and local parity blocks because they require the same number of blocks for repair.

Code	NRC	SSD	Opt HDD	Cold HDD
Reed-Solomon	15	100	115	303
Azure-LRC	6.6 (0.44)	58 (0.58)	65 (0.56)	134 (0.44)
Azure-LRC+1	4.8 (0.32)	49 (0.49)	53 (0.46)	134 (0.44)
Optimal-LRC	6 (0.4)	54 (0.54)	57 (0.49)	143 (0.47)

Table 2: NRC of all codes and their recovery time in seconds. $n = 15$ and $k = 10$ for all codes and $r = 4$ for the LRCs, with the repair time normalized to Reed-Solomon in parentheses.

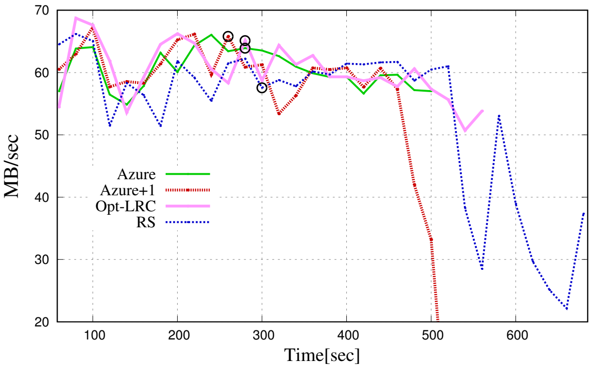


Figure 15: Throughput of RADOS benchmark during repair with LRC in (15,10,4) and RS(15,10).

tion in repair time.

Different storage types. LRCs reduce the amount of data read during recovery, and thus their benefit is expected to increase with the cost of storage I/O. We repeated our repair experiment for one configuration, replacing the SSD storage volumes with two types of hard drives, Optimized HDD and Cold HDD, with a maximum throughput of 500 and 250 IOPS, respectively [1]. The amount of data read from all storage types was the same. Table 2 shows the repair time, in seconds, for all the codes and storage types. As we expected, in the setups where the repair time of Reed-Solomon was longer, the reduction in repair time achieved by all LRCs was higher and closer to the reduction predicted by NRC.

Foreground workloads. Local repair is also designed to minimize the interference with application workloads running in the system at the time of failure. To evaluate this interference, we repeated the repair experiment with the (15,10,4) configuration, in which each LRC has a different NRC. We ran a Ceph benchmark called RADOS Bench [32], which writes objects for a given amount of time (220 seconds in our experiment), reads all the objects, and terminates. For this experiment, we increased the number of outstanding recovery requests allowed per OSD from 15 to 150. We killed one OSD 100 seconds after the benchmark started to read. The repair process took place while the benchmark was still reading the data, but the system recovered before the benchmark terminated.

Figure 15 shows the throughput of the benchmark's I/O requests during its read phase. The black circles mark the time at which recovery was fully completed, and the measurements continue until the benchmark terminates. The differences between the codes were smaller than we expected. This is the result of Ceph's restrictions on repair throughput, and of the high I/O parallelism of SSDs. Nevertheless, the results show that the different codes completed their repair in the order of their NRC: Azure-LRC+1 was the fastest and Reed-Solomon the slowest. The throughput reduction experienced by the benchmark was greatest with Reed-Solomon and smallest with the full-LRCs—Azure-LRC+1 and Optimal-LRC.

Multiple zones. The first LRCs were motivated by the goal of restricting the repair cost to the locality of the failed node. In production systems, this means that blocks in the same group are assigned to a group of nodes on the same rack or in the same zone of the datacenter [11]. To evaluate the different LRCs in a similar environment, we repeated the repair experiment when our instances were deployed on three availability zones in the same EC2 region (N. Virginia). In this experiment, we deployed six instances in each zone, with a total of 18 instances running 36 OSDs. To make up for the reduced I/O bandwidth within in each zone, we replaced the General Purpose SSDs with Provisioned IOPS SSDs, which increased the maximum IOPS per volume from 150 to 2500.

We edited the CRUSH map, instructing CRUSH to assign placement groups to OSDs such that groups are allocated in the same zone [35]. We also ensured that the primary OSD resides in the same zone as the failed OSD. We ran this experiment twice. In the first setup, both the primary OSD and the failed OSD belonged to a data group. In the second setup, both the primary OSD and the failed OSD belonged to a global-parity group.

For full-LRCs, both setups are equivalent: all blocks are reconstructed from blocks in the same group. For Reed-Solomon, all recovery scenarios follow the second setup: recovery requires blocks from different groups. For data-LRCs, data and local parity blocks are recovered according to the first setup, and global parities are recovered according to the second setup. We calculated the weighted average of these two setups to obtain the expected recovery time for each code.

We used (15,8,4) as our configuration, having excluded it from our previous analysis due to its high overhead. Nevertheless, it has the desirable property that all the groups in all codes have the same size (5). This ensures that all placement groups include the same number of OSDs in each zone. In this configuration, the full LRCs are equivalent in their distribution of data and parity blocks to groups. We use Azure-LRC+1 as our full-LRC in this experiment.

For comparison, we repeated this experiment with all OSDs in a single zone, but with the same restriction on the allocation of OSDs to groups. This setup eliminates the cross-zone network bottleneck, with I/O parallelism limited as in the three-zone experiment. Our baseline is

Code	NRC	Reads	1 zone		3 zones
			Baseline	3 groups	
Reed-Solomon	15	14.42	121	179	190
Azure-LRC	10	9.73	88 (0.73)	158 (0.88)	162 (0.85)
Azure-LRC+1	7.5	7.21	80 (0.66)	148 (0.82)	148 (0.78)

Table 3: Number of blocks read per lost data block and repair time when running on one zone and on three zones, with the repair time normalized to Reed-Solomon in parentheses.

the unrestricted setup we used in the rest of this section.

Table 3 shows the amount of data read and the weighted average of the repair time in this experiment. It shows that restricting the number of nodes that participate in the repair process significantly reduces its throughput. When all the OSDs are deployed in the same zone, this restriction increases the repair time by 48% to 85%. The increase is lower for Reed-Solomon because it can still utilize twice as many OSDs than the LRCs. The addition of the cross-zone network bottleneck further reduces the repair time of Reed-Solomon (by 6%) and of Azure-LRC (by 2.5%), but does not affect Azure-LRC+1 which does not incur any cross-zone transfers for repair.

These results demonstrate the well-known tradeoff between I/O parallelism and locality. They confirm that data-LRCs and full-LRCs are expected to achieve the highest benefit in large-scale deployments, where sufficient I/O parallelism can be achieved within a single zone.

7 Related Work

Efforts to reduce the repair cost can be classified into two main lines of research: LRCs, which attempt to reduce the repair cost by reducing the number of nodes participating in the repair process [8, 10, 11, 17, 28], and Regenerating Codes, which strive to attain the same goal by reducing the network bandwidth utilized during repair [6, 24, 25].

The benefits of codes with locality were first realized in [10] before the actual notion was isolated into a standalone concept in the information-theory community [8]. The basic code construction of Pyramid codes [10] assumes that a global parity relation of an MDS code is subdivided into two (or more) local parity check equations which can be used for local repair. Importantly, this work indicated possible savings in repair cost, thereby propelling further research on LRC codes.

In particular, [11] developed LRC codes and observed substantial savings in the repair cost of Microsoft Azure storage attained by using them, and [8] developed the coding-theoretic side of the notion of LRCs. Another relevant work [37] builds on Azure-LRC to dynamically adjust the system's overall storage overhead and average recovery speed by migrating hot and cold data to arrays with more or fewer parity nodes, respectively. Finally, a family of non-MDS codes called Sector Disk codes [15, 19] addresses the recovery of a failed block within an otherwise healthy node. The codes constructed in these works add parity blocks that allow efficient recovery of bad hard-disk sectors or SSD blocks. The above codes were implemented and evaluated independently of one another. Our study is the first to present a comparative framework for codes designed with different properties and overheads, and for different sets of parameters.

Minimum storage regenerating (MSR) codes [6,25] are a class of MDS codes designed to optimize recovery network bandwidth rather than the number of accessed storage devices. MSR codes and related families, such as RotatedRS [12], Hitchhiker-XOR [23], Butterfly [7] and Zigzag [31] codes, divide each data and parity block into smaller chunks, such that only a subset of each block's chunks are required for the repair of a failed node. Paper [22] constructs an MSR code that reduces the amount of data read from some of the surviving nodes but is applicable only for clusters with $n = 2 \times k$. These codes reduce the *rebuilding ratio*—the portion of the surviving nodes' data that must be read during recovery. All MDS codes with the same d have the same overhead, and can be directly compared by their rebuilding ratio. However, this metric is also limited in its ability to predict recovery costs in a real system: these costs depend on the granularity of the non-sequential I/O accesses incurred when reading arbitrary chunks from each block [18].

Alternative approaches reduce recovery costs of existing codes. An approach introduced in [33] and developed in [9] considers linear repair schemes of Reed-Solomon codes for reducing their network repair bandwidth. *Repair pipelining* improves the utilization of the network bandwidth of all nodes participating in the recovery [16]. *Lazy repair* delays node recovery to amortize its costs over more than one failure [29]. This reduces the fault tolerance of the system, which is equivalent to reducing d. LSTOR relies on attached non-volatile memory for caching additional parity blocks [27], effectively increasing the storage overhead of the system. Our comparative framework, using NRC, can also be extended to evaluate the above approaches.

8 Conclusions

In this study, we performed the first systematic comparison of full-LRCs and data-LRCs. To that end, we implemented a new full-LRC, Azure-LRC+1, and extended and implemented Optimal-LRC for a comparison covering a wide range of system parameters. We demonstrated the limitations of existing metrics and introduced NRC—a new metric that successfully models full-node repair cost. Our theoretical analysis demonstrated the non-trivial correlation between NRC and the cost of degraded reads, and the tradeoff between them and the code's fault tolerance. Our evaluation in a Ceph cluster on Amazon EC2 further showed how this benefit of full-LRCs and data-LRCs depends on the underlying storage devices, network topology, and foreground application load.

References

[1] Amazon EBS volumes. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumes.html, 2017. [Accessed: 2017-09-24].

[2] Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types, 2017. [Accessed: 2017-09-22].

[3] Amazon EC2 regions and availability zones. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html, 2017. [Accessed: 2017-09-22].

[4] Jerasure erasure code plugin. http://docs.ceph.com/docs/hammer/rados/operations/erasure-code-jerasure/, 2017. [Accessed: 2017-09-24].

[5] Locally repairable erasure code plugin. http://docs.ceph.com/docs/hammer/rados/operations/erasure-code-lrc/, 2017. [Accessed: 2017-09-24].

[6] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, Sept 2010.

[7] E. En-Gad, R. Mateescu, F. Blagojevic, C. Guyot, and Z. Bandic. Repair-optimal MDS array codes over $GF(2)$. In *IEEE International Symposium on Information Theory (ISIT)*, 2013.

[8] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin. On the locality of codeword symbols. *IEEE Transactions on Information Theory*, 58(11):6925–6934, November 2012.

[9] V. Guruswami and M. Wootters. Repairing reed-solomon codes. In *48th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, 2016.

[10] C. Huang, M. Chen, and J. Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *Trans. Storage*, 9(1):3:1–3:28, Mar. 2013.

[11] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX Annual Technical Conference (ATC)*, 2012.

[12] O. Khan, R. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *10th Usenix Conference on File and Storage Technologies (FAST)*, 2012.

[13] O. Kolosov, A. Barg, I. Tamo, and G. Yadgar. Optimal LRC codes for all lenghts n $\leq$ q. *ArXiv e-prints*, abs/1802.00157, Feb. 2018.

[14] J. Li and X. Tang. Optimal exact repair strategy for the parity nodes of the $(k+2,k)$ zigzag code. *IEEE Transactions on Information Theory*, 62(9):4848–4856, Sept 2016.

[15] M. Li and P. P. C. Lee. STAIR codes: A general family of erasure codes for tolerating device and sector failures. *Trans. Storage*, 10(4):14:1–14:30, Oct. 2014.

[16] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair pipelining for erasure-coded storage. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[17] F. Oggier and A. Datta. Self-repairing homomorphic codes for distributed storage systems. In *Proc. 2011 IEEE INFOCOM*, pages 1215–1223, 2011.

[18] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Guyot, E. En-Gad, and Z. Bandic. Opening the chrysalis: On the real repair performance of MSR codes. In *14th Usenix Conference on File and Storage Technologies (FAST)*, 2016.

[19] J. S. Plank and M. Blaum. Sector-disk (SD) erasure codes for mixed failure modes in RAID systems. *Trans. Storage*, 10(1):4:1–4:17, Jan. 2014.

[20] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois field arithmetic using Intel SIMD instructions. In *11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[21] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *7th Usenix Conference on File and Storage Technologies (FAST)*, 2009.

[22] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[23] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *ACM SIGCOMM*, 2014.

[24] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to

the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[25] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction. *IEEE Transactions on Information Theory*, 57(8):5227–5239, Aug 2011.

[26] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[27] E. Rosenfeld, N. Amit, and D. Tsafrir. Using disk add-ons to withstand simultaneous disk failures with fewer replicas. *7th Annual Workshop on the Interaction amongst Virtualization, Operating Systems and Computer Architecture (WIVOSCA)*, 2013.

[28] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: novel erasure codes for big data. In *39th International Conference on Very Large Data Bases (VLDB)*, 2013.

[29] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *International Conference on Systems and Storage (SYSTOR)*, 2014.

[30] I. Tamo and A. Barg. A family of optimal locally recoverable codes. *IEEE Transactions on Information Theory*, 60(8):4661–4676, Aug 2014.

[31] I. Tamo, Z. Wang, and J. Bruck. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Transactions on Information Theory*, 59(3):1597–1616, March 2013.

[32] F. Wang, M. Nelson, S. Oral, S. Atchley, S. Weil, B. W. Settlemyer, B. Caldwell, and J. Hill. Performance and scalability evaluation of the ceph parallel file system. In *Proceedings of the 8th Parallel Data Storage Workshop. ACM*, 2013.

[33] Z. Wang, A. Dimakis, , and J. Bruck. Rebuilding for array codes in distributed storage systems. In *GLOBECOM Workshops (GC Wkshps)*, pages 1905–1909. IEEE, 2010.

[34] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[35] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *ACM/IEEE Conference on Supercomputing (SC)*, 2006.

[36] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. RADOS: A scalable, reliable storage service for petabyte-scale storage clusters. In *2nd International Workshop on Petascale Data Storage (PDSW): Held in Conjunction with Supercomputing*, 2007.

[37] M. Xia, M. Saxena, M. Blaum, and D. A. Pease. A tale of two erasure codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[38] M. Ye and A. Barg. Explicit constructions of MDS array codes and RS codes with optimal repair bandwidth. In *2016 IEEE International Symposium on Information Theory (ISIT)*, 2016.

TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions

Yige Hu[1], Zhiting Zhu[1], Ian Neal[1], Youngjin Kwon[1], Tianyu Cheng[1]
Vijay Chidambaram[1,2], Emmett Witchel[1]
[1]*University of Texas at Austin* [2]*VMware Research*

Abstract

We introduce TxFS, a novel transactional file system that builds upon a file system's atomic-update mechanism such as journaling. Though prior work has explored a number of transactional file systems, TxFS has a unique set of properties: a simple API, portability across different hardware, high performance, low complexity (by building on the journal), and full ACID transactions. We port SQLite and Git to use TxFS, and experimentally show that TxFS provides strong crash consistency while providing equal or better performance.

1 Introduction

Modern applications store persistent state across multiple files [21]. Some applications split their state among embedded databases, key-value stores, and file systems [27]. Such applications need to ensure that their data is not corrupted or lost in the event of a crash. Unfortunately, existing techniques for crash consistency, such as logging or using atomic rename, result in complex protocols and subtle bugs [21].

Transactions present an intuitive way to atomically update persistent state [6]. Unfortunately, building transactional systems is complex and error-prone. In this paper, we introduce a novel approach to building a transactional file system. We take advantage of a mature, well-tested piece of functionality in the operating system: the file-system journal, which is used to ensure atomic updates to the internal state of the file system. We use the atomicity and durability provided by journal transactions and leverage it to build ACID transactions available to user-space transactions. Our approach greatly reduces the development effort and complexity for building a transactional file system.

We introduce TxFS, a transactional file system that builds on the ext4 file system's journaling mechanism. We designed TxFS to be practical to implement and to use. TxFS has a unique set of properties: it has a small implementation (5,200 LOC) by building on the journal (for example, TxFS has 25% the LOC of the TxOS transactional operating system [22]); it provides high performance unlike various solutions which built a transactional file system over a user-space database [5, 16, 18, 31]; it has a simple API (just wrap code in `fs_tx_begin()` and `fs_tx_commit()`) compared to solutions like Valor [28] or TxF [24] which require multiple system calls per transaction and can require the developer to understand implementation details like logging; it provides all ACID guarantees unlike solutions such as CFS [15] and AdvFS [30] which only offer some of the guarantees; it provides transactions at the file level instead of at the block level unlike Isotope [26], making several optimizations easier to implement; finally, TxFS does not depend upon specific properties of the underlying storage unlike solutions such as MARS [3] and TxFlash [23].

The advantage to building TxFS on the file-system journal is that TxFS transactions obtain atomicity, consistency, and durability by placing each one entirely within a single file-system journal transaction (which is applied atomically to the file system). Using well-tested journal code to obtain ACD reduces the implementation complexity of TxFS, while limiting the maximum size of transactions to the size of the journal.

The main challenge of building TxFS is providing isolation. Isolation for TxFS transactions requires that in-progress TxFS transactions are not visible to other processes until the transaction commits. At a high level, TxFS achieves isolation by making private copies of all data that is read or written, and updating global data during commit. However, the naive implementation of this approach would be extremely inefficient: global data structures such as bitmaps would cause conflicts for every transaction, causing high abort rates and excessive transaction retries. TxFS makes concurrent transactions efficient by collecting logical updates to global structures, and applying the updates at commit time. TxFS includes a number of other optimizations such as eager conflict detection that are tailored to the current implementation of file-system data structures in ext4.

We find that the transactional framework allows us to easily implement a number of file-system optimizations. For example, one of the core techniques from our earlier work, separating ordering from durability [2], is easily accomplished in TxFS. Similarly, we find TxFS transactions allow us to identify and eliminate redundant application IO where temporary files or logs are used to atomically update a file: when the sequence is simply enclosed in a transaction (and without any other changes), TxFS atomically updates the file (maintaining functionality) while eliminating the IO to logs or temporary files (provided the temporary files and logs are deleted inside the transaction). As a result, TxFS improves performance while simultaneously providing better crash-consistency semantics: a crash does not leave ugly temporary files or logs that need to be cleaned up.

To demonstrate the power and ease of use of TxFS transactions, we modify SQLite and Git to incorporate TxFS transactions. We show that when using TxFS transactions, SQLite performance on the TPC-C benchmark improves by 1.6× and a micro-benchmark which mimics Android Mail obtains 2.3× better throughput. Using TxFS transactions greatly simplifies Git's code while providing crash consistency without performance overhead. Thus, TxFS transactions increase performance, reduce complexity, and provide crash consistency.

Our paper makes the following contributions.

- We present the design and implementation of TxFS, a transactional file system for modern applications built by leveraging the file-system journal (§3). We have made TxFS publicly available at `https://github.com/ut-osa/txfs`.
- We show that existing file systems optimizations, such as separating ordering from durability, can be effectively implemented for TxFS transactions (§4).
- We show that real applications can be easily modified to use TxFS, resulting in better crash semantics and significantly increased performance (§5).

2 Background and motivation

We first describe the protocols used by current applications to update state in a crash-consistent manner. We then present a study of different applications and the challenges they face in maintaining crash consistency across persistent state stored in different abstractions. We describe the file-system optimizations enabled by transactions and finally summarize why we think transactional file systems should be revisited.

2.1 How applications update state today

Given that applications today do not have access to transactions, how do they consistently update state to multiple storage locations? Even if the system crashes or power fails, applications need to maintain invariants across state in different files (*e.g.,* an image file should match the thumbnail in a picture Gallery). Applications achieve this by using ad hoc protocols that are complex and error-prone [21].

In this section, we show how difficult it is to implement seemingly simple protocols for consistent updates to storage. There are many details that are often overlooked, like the persistence of directory contents. These protocols are complex, error prone, and inefficient. With current storage technologies, these protocols must sacrifice performance to be correct because there is no efficient way to order storage updates.

```
open(/dir/tmp)
write(/dir/tmp)
fsync(/dir/tmp)
fsync(/dir)
rename(/dir/tmp, /dir/orig)
fsync(/dir/)
```

(a) Atomic Update via Rename

```
open(/dir/log)
write(/dir/log)
fsync(/dir/log)
fsync(/dir/)
write(/dir/orig)
fsync(/dir/orig)
unlink(/dir/log)
fsync(/dir/)
```

(b) Atomic Update via Logging

```
// Write attachment
open(/dir/attachment)
write(/dir/attachment)
fsync(/dir/attachment)
fsync(/dir/)

// Writing SQLite Database
open(/dir/journal)
write(/dir/journal)
fsync(/dir/journal)
fsync(/dir/)
write(/dir/db)
fsync(/dir/db)
unlink(/dir/journal)
fsync(/dir/)
```

(c) Atomically adding a email message with attachments in Android Mail

Figure 1: Different protocols used by applications to make consistent updates to persistent data.

Currently, applications use the `fsync()` system call to order updates to storage [2]; since `fsync()` forces durability of data, the latency of a `fsync()` call varies from a few milliseconds to several seconds. As a result, applications do not call `fsync()` at all the places in the update protocol where it is necessary, leading to severe data loss and corruption bugs [21].

We now describe two common techniques used by applications to consistently update stable storage. Figure 1 illustrates these protocols.

Atomic rename. Protocol (a) shows how a file can be updated via atomic rename. The atomic rename approach is widely used by editors, such as Emacs and Vim, and by GNOME applications that need to atomically update dot configuration files. The application writes new data to a temporary file, persists it with an `fsync()` call, updates the parent directory with another `fsync()` call, and then renames the temporary file *over* the original file, effectively causing the directory entry of the original file to point to the temporary file instead. The old contents of the original file are unlinked and deleted. Finally, to ensure that the temporary file has been unlinked properly, the application calls `fsync()` on the parent directory.

Logging. Protocol (b) shows another popular technique for atomic updates, logging [8] (either write-ahead-logging or undo logging). The log file is written with new contents, and both the log file and the parent directory (with the new pointer to log file) are persisted. The application then updates the original file and persists the original file; the parent directory does not change during this step. Finally, the log is unlinked, and the parent directory is persisted.

The situation becomes more complex when applications store state across multiple files. Protocol (c) illustrates how the Android Mail application adds a new email with an attachment. The attachment is stored on the file system, while the email message (along with metadata) is stored in the database (which for SQLite, also resides on the file system). Since the database has a pointer to the attachment (i.e., a file name), the attachment must be persisted first. Persisting the attachment requires two `fsync()` calls (to the file and its containing directory) [1, 21]. SQLite's most performant mode uses write-ahead-logging to atomically update the database. It then follows a protocol similar to Protocol (b).

Removing `fsync()` calls in any of the presented protocols will lead to data loss or corruption. For instance, in Protocol (b), if the parent directory is not persisted with an `fsync()` call, the following scenario could occur: the application writes the log file, and then starts overwriting the original file in place. The system crashes at this point. Upon reboot, the log file does not exist, since the directory entry pointing to the log file was not persisted. Thus, the application file has been irreversibly partially edited, and cannot be restored to a consistent version. Many application developers avoid `fsync()` calls due to the resulting decrease in performance, leading to severe bugs that cause loss of data.

Safe update protocols for stable storage are complex and low performance (*e.g.,* Android Mail uses *six* `fsync()` calls to persist a single email with an attachment). System support for transactions will provide high performance for these applications.

2.2 Application case studies

We now present four examples of applications that struggle with obtaining crash consistency using primitives available today. Several applications store data across the file system, key-value stores, and embedded databases such as SQLite [27]. While all of this data ultimately resides in the file system, their APIs and performance constraints are different and consistently updating state across these systems is complex and error-prone.

Android mail. Android's default mail application stores mail messages using the SQLite embedded database [29]. Mail attachments are stored separately as a file, and the database stores a pointer to the file. The user requires both the file and the database to be updated atomically; SQLite only ensures the database is updated correctly. For example, a crash could leave the database consistent, but with a dangling pointer to a missing attachment file. The mail application handles this by first persisting the attachment (via `fsync()`), and then persisting a database transaction. Clearly, this harms performance – a transaction that spans both the database and the file system would need to persist data only at a single commit point.

Apple iWork and iLife. Analysis of the storage behavior of Apple's home-user, desktop applications [9] finds that applications use a combination of the file system, key-value stores, and SQLite to store data. iTunes uses SQLite to store metadata similar to the Android Mail application. When you download a new song via iTunes, the sound file is transferred and the database updated with the song's metadata. Apple's Pages application uses a combination of SQLite and key-value stores for user preferences and other metadata (two SQLite databases and 128 `.plist` key-value store files). Similar to Android Mail, Apple uses `fsync()` to order updates correctly.

Browsers. Mozilla Firefox stores user data in multiple SQLite databases. For example, addons, cookies, and download history are each stored in their separate SQLite database. Since downloads and other files are stored on the file system, a crash could leave a database with a dangling pointer to a missing file.

Version control systems. Git and Mercurial are widely-used version control systems. The `git commit` command requires two file-system operations to be atomic: a file append (`logs/HEAD`) and a file rename (to a lock file). Failure to achieve atomicity results in data loss and a corrupted repository [21]. Mercurial uses a combination of different files (`journal, filelog, manifest`) to consistently update state. Mercurial's `commit` command requires a long sequence of file-system operations including file creations, appends, and renames be atomic; if not, the repository is corrupted [21].

For these applications, transactional support would lead directly to more understandable and more efficient idioms. It is difficult for a user-level program to provide crash-consistent transactional updates using the POSIX file-system interface. A transactional file-system interface will also enable high-performance idioms like editors grouping updates into transactions rather than the less efficient process they currently use of making temporary file copies that are committed via `rename`.

Note that applications that use temporary files and techniques like atomic rename do achieve crash consistency; however, after a crash there may be temporary files which need to be cleaned up. After a crash, the application runs a recovery procedure and returns to a consistent state. Often, the "recovery procedure" forces a human user to look for and manually delete stale files. A transactional file system does not provide *new* crash-consistency guarantees for these applications; rather, transactional file systems remove the burden of recovery and cleanup, simplifying the application and eliminating bugs [21].

2.3 Optimizing transactions

A transactional file-system interface enables a number of interesting file-system optimizations. We now describe a few of them.

Eliminate temporary durable files. A number of applications such as Vim, Emacs, Git, and LevelDB provide reasonable crash semantics (*i.e.,* the user sees either the old version or the new version after an update) by making a temporary copy of a file, editing it, then renaming it atomically to the permanent name when the user updates data. The application can simply enclose its writes inside a transaction, avoiding the copy. For large files, the difference in performance can be significant. In addition, the file system will not be cluttered with temporary files in the event of a crash.

Group commit. Transactions buffer related file-system updates in memory, which can all be sent to the storage device at once. Batching updates is often more efficient, enabling efficient allocation of file-system data structures and better device-level scheduling. Without user-provided transaction boundaries, the file system provides uniform, best-effort persistence for all updates.

Eliminate redundant IO *within* transactions. Workloads often contain redundancy; for example, files are often updated several times at the same offset, or a file is created, written, read, and unlinked. Transaction boundaries allow the file system to eliminate some of this redundant work because the entire transaction is visible to the file system at commit time, which enables global optimization.

Consolidate IO *across* transactions. Transactions often update data written by prior transactions. When a workload anticipates data in its transaction will be updated by another transaction shortly, it can prioritize throughput over latency. Committing a transaction with a special flag allows the system to delay a transaction commit, anticipating that the data will be overwritten, and then it can be persisted once instead of twice. Note that consolidating IO in this manner is different from eliminating redundant IO within a transaction; this optimization operates across multiple transactions. Optimizing multiple transactions, especially from different applications, is best done by the operating system, not by an individual application. This non-work conserving strategy is similar to the anticipatory disk scheduler [12].

Separate ordering from durability. When ending a transaction, the programmer can specify if the transaction should commit durably. If so, the call blocks until all updates specified by the transaction have been written to a persistent journal. If we commit non-durable transaction A and then start non-durable transaction B, then A is ordered before B, but neither is durable. A subsequent transaction (*e.g.,* C), can specify that it and all previous transactions should be made durable. In this way we can use transactions to gain much of the benefit of splitting `sync` into ordering sync (`osync`), and durability sync (`dsync`) [2].

In summary, we believe transactional file systems should be revisited for two reasons. First, applications routinely store persistent state in multiple files and across different storage systems such as databases and key-value stores, and maintaining crash consistency of this state using techniques such as atomic rename results in complexity and bugs. Second, using a transactional API enables the file system to provide a number of optimizations that would be significantly harder to introduce in a non-transactional file system.

3 TxFS Design and implementation

We now present the design and implementation of TxFS. TxFS avoids the pitfalls from earlier transactional file systems (§6): it has a simple API; provides complete ACID guarantees; does not depend on specific hardware; and takes advantage of the file-system journal and how the kernel is implemented to achieve a small implementation (≈5,200 LOC).

3.1 API

A simple API was one of the key goals of TxFS. Thus, TxFS provides developers with only three system calls: `fs_tx_begin()`, which begins a transaction; `fs_tx_commit()`, which ends a transaction and attempts to commit it; and `fs_tx_abort()`, which discards all file-system updates contained in the current transaction. On commit, all file-system updates in an application-level transaction are persisted in an atomic fashion – after a crash, users see all of the transaction updates, or none of them. This API significantly simplifies application code and provides clean crash semantics, since temporary files or partially written logs will not need to be cleaned up after a crash.

`fs_tx_commit()` returns a value indicating whether the transaction was committed successfully, or if it failed, why it failed. A transaction can fail for three reasons: there was a conflict with another concurrent transaction, there is no journal space for the transaction, or the file system does not have enough resources for the transaction to complete (no space or inodes). Depending upon the error code, the application can choose to retry the transaction. Nested TxFS transactions are flattened into a single transaction, which succeed or fail as a unit. Flat nesting is a common choice in transactional systems [22, 28].

A user can surround any sequence of file-system related system calls with `fs_tx_begin()` and `fs_tx_commit()` and the system will execute those system calls in a single transaction. This interface is easy for programmers to use and makes it simple to incrementally deploy file-system transactions into existing applications. In contrast, some transactional file systems (*e.g.,* Window's TxF [24] and Valor [28]) have far more complex, difficult-to-use interfaces. TxF assigns a handle to each transaction, and requires users to explicitly call the transactional APIs with the handle. Valor exposes operations on the kernel log to user-level code.

TxFS isolates file-system updates only. The application is still responsible for synchronizing access to its own user-level data structures. A transactional file system is not intended to be an application's sole concurrency control mechanism; it only coordinates file-system updates which are difficult to coordinate without transactions.

3.2 Atomicity and durability

Most modern Linux file systems have an internal mechanism for atomically updating multiple blocks on storage. These mechanisms are crucial for maintaining file-system crash consistency, and thus have well-tested and mature implementations. TxFS takes advantage of these mechanisms to obtain three of the ACID properties: atomicity, consistency, and durability. This is the key insight which allows TxFS to have a small implementation.

TxFS builds upon the ext4 file system's journal. The journal provides the guarantee that each journal transaction is applied to the file system in an atomic fashion. We could have instead used a different mechanism such as copy-on-write [10] which provides the same guarantee in btrfs and F2FS. TxFS can be built upon any file system with a mechanism for atomic updates.

For each TxFS transaction, TxFS maintains a private `jbd2` transaction, and at commit, merges the private transaction into the global `jbd2` transaction. While the global `jbd2` transaction contains only metadata by default, TxFS also adds data blocks to the transaction to ensure atomic updates. If, by chance, a block added to the private `jbd2` transaction is also being committed by a previous global `jbd2` transaction, TxFS creates a shadow block. Ext4 also creates a shadow block when a block is shared between a running and a committing transaction. TxFS employs selective data journaling [2], only journaling data blocks that were already allocated (*i.e.,* data blocks that are being updated), and avoids journaling newly allocated data blocks (because it can write them directly). Selective data journaling provides the same guarantees as full data journaling at a fraction of the cost.

TxFS ensures that an entire transaction can be merged into a *single* journal transaction; otherwise, an error is returned to the user. As long as a TxFS transaction is added to a single journal transaction, the journal will ensure it is applied to the file system atomically. After merging a user's transaction into the journal transaction, TxFS persists the journal transaction, ensuring the durability of the TxFS transaction.

3.3 Isolation and conflict detection

Although the ext4 journal provides atomicity and durability, it does not provide isolation. Adding isolation for file-system data structures in the Linux kernel is challenging because a large number of functions all over the kernel modify file-system data structures without using a common interface. In TxFS, we tailor our approach to isolation for each data structure to simplify the implementation.

To provide isolation, TxFS has to ensure that all operations performed inside a transaction are not visible to other transactions or the rest of the system until commit time. TxFS achieves the isolation level of *repeatable reads* [7] using a combination of different techniques.

Split file-system functions. System calls such as `write()` and `open()` execute file-system functions which often result in allocation of file-system resources such as data blocks and inodes. TxFS splits such functions into two parts: one part which does file-system allocation, and one part which operates on in-memory structures. The part doing file-system allocation is moved to the commit point. The other part is executed as part of the system call, and the in-memory changes are kept private to the transaction.

Transaction-private copies. TxFS makes transaction-private copies of all kernel data structures modified during the transaction. File-system related system calls inside a transaction operate on these private copies, allowing transactions to read their own writes. In case of abort, these private copies are discarded; in case of commit, these private copies are carefully applied to the global state of the file system in an atomic fashion. During a transaction, file-system operations are redirected to the local in-memory versions of the data structures. For example, dentries updated by the transaction are modified to point to a local inode which maintains a local radix tree which has locally modified pages.

Two phase commit. TxFS transactions are committed using a two-phase commit protocol. TxFS first obtains a lock on all relevant file-system data structures using a total order. The following order prevents deadlock: inode mutexes, page locks, inode buffer head locks, the global `inode_hash_lock`, the global `inode_sb_list_lock`, inode locks, and dentry locks. The Linux kernel orders the acquiring of inode mutexes based on the pointer addresses of their inodes; we adopt this locking discipline in TxFS. Similarly, page locks are acquired in order of the address of the page. Acquiring the locks for directory data block buffers and inode metadata buffers is ordered by inode number.

After obtaining the locks, all allocation decisions are checked to see if they would succeed; for example, if the transaction creates inodes, TxFS checks if there are enough free inodes. Next, TxFS checks the journal to ensure there is enough space in the global `jbd2` transaction to allow the transaction to be merged. Finally, TxFS

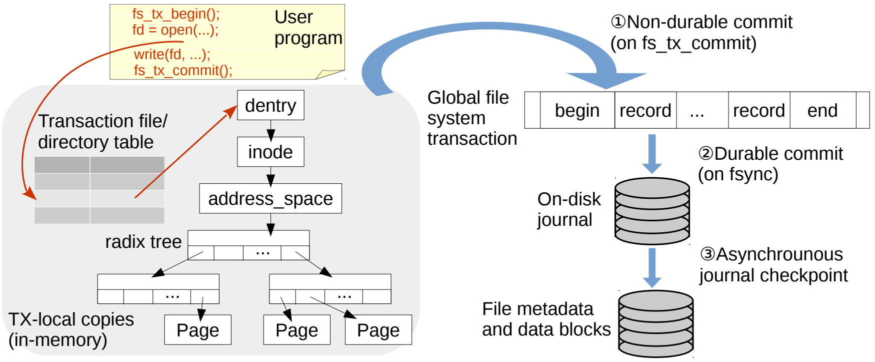

```
fs_tx_begin();
fd = open(...);
write(fd, ...);
fs_tx_commit();
```
User program
Transaction file/ directory table
dentry
inode
address_space
radix tree
...
...
...
TX-local copies (in-memory)
Page
Page
Page
①Non-durable commit (on fs_tx_commit)
Global file system transaction
begin
record
...
record
end
②Durable commit (on fsync)
On-disk journal
③Asynchrounous journal checkpoint
File metadata and data blocks

Figure 2: TxFS relies on ext4's own journal for atomic updates and maintains local copies of in-memory data structures, such as inodes, dentries, and pages to provide isolation guarantees. At commit time, the local operations are made global and durable.

checks for conflicts with other transactions (as described below). If any of these checks fail, all locks are released, and the commit returns an error to the user. Otherwise, the in-memory data structures are updated, all file-system allocation is performed, and the private `jbd2` transaction is merged with the global `jbd2` transaction. At this point, the transaction is committed, locks are released and the changes are persisted to storage in a crash-consistent manner.

Conflict detection. Conflict detection is a key part of providing isolation. Since allocation structures such as bitmaps are not modified until commit time, they cannot be modified by multiple transactions at the same time, and do not give rise to conflicts; as a result, TxFS avoids false conflicts involving global allocation structures.

Conflict detection is challenging as file-system data structures are modified all over the Linux kernel without a standard interface. TxFS takes advantage of how file-system data structures are implemented to detect conflicts efficiently.

Conflict detection for pages. The `struct page` data structure holds the data for cached files. TxFS adds two fields to this structure: `write_flag` and `reader_count`. The `write_flag` indicates if there is another transaction that has written this page. The `reader_count` field indicates the number of other transaction that have read this page. Non-transactional threads will never see the in-flight un-committed data in transactions, and thus can always safely read data. TxFS does eager conflict detection for pages since there is a single interface to read and write pages that TxFS interposes. The following rules are followed on a page read or write:

1. When a transaction reads a page, it increments `reader_count` by one. If the page has the `write_flag` set, the transaction aborts.
2. If a transaction attempts to write a page that has either the `write_flag` set or `reader_count` greater than zero, it aborts. Otherwise, it sets the `write_flag`.
3. If a non-transactional thread attempts to write to a page with `reader_count` or `write_flag` set, it is put to sleep until the transaction commits or aborts.
4. When the transaction commits or aborts, `write_flag` is reset and `reader_count` is decremented.

Aborting transactions in this manner can lead to livelock, but we have not found it a problem with our benchmarks and the policy can be easily changed to resolve conflicts in favor of the oldest transaction (which does not livelock). TxFS favors transactional throughput, but for greater fairness between transactional and non-transactional threads, TxFS could allow a non-transactional thread to proceed by aborting all transactions conflicted by its operation [22].

Conflict detection for dentries and inodes. Apart from pages, TxFS must detect conflicts on two other data structures: dentries (directory entries) and inodes. Unfortunately, unlike pages, inodes and dentries do not have a standard interface and are modified throughout kernel code. Therefore, TxFS uses lazy conflict detection for inodes and dentries, detecting conflicts at commit time. At commit time, TxFS needs to detect if the global copy of the data structure has changed since it was copied into the local transaction. Doing a byte-by-byte comparison of each modified data structure would significantly increase commit latency; instead, TxFS takes advantage of the inode's `i_ctime` field that is changed whenever the inode is changed; TxFS simply has to check that the `i_ctime` has not changed for each inode that TxFS has read or written (writes are performed to a transaction-local copy of the inode). TxFS similarly adds a new `d_ctime` field to the dentry data structure to track its last modified time. We added kernel code in a number of places to update `d_ctime` whenever a dentry is changed. Creating different named entries within a directory does not create a conflict because the names are checked at commit time. By taking advantage of `i_ctime` and `d_ctime`, TxFS is able to perform conflict detection for these structures without radically changing the Linux kernel.

Summary. Figure 2 shows how TxFS uses ext4's journal for atomically updating operations inside a transaction, and maintaining local state to provide isolation guarantees. File operations inside a TxFS transaction are redirected to the transaction's local copied data structures, hence they do not affect the file system's global state, while being observable by subsequent operations in the same transaction. Only after a TxFS transaction finishes its commit (by calling `fs_tx_commit()`) will its modifications be globally visible.

3.4 Implementation

We modified Linux version 3.18 and the ext4 file system. The implementation requires a total of 5,200 lines of code, with 1,300 in TxFS internal bookkeeping, 1,600 in the VFS layer, 900 in the journal (JBD2) layer, 1,200 for ext4 and 200 for memory management (all measurements with SLOCCount [4]). Except for the ext4 and jbd2 extensions, all other code could be reused to port TxFS to other file systems, such as ZFS, in the future.

3.5 Limitations

TxFS has two main limitations. First, the maximum size of a TxFS transaction is limited to one fourth the size of the journal (the maximum journal transaction size allowed by ext4). We note that the journal can be configured to be as large as required. Multi-gigabyte journals are common today. Second, although parallel transactions can proceed with ACID guarantees, each transaction can only contain operations from a single process. Transactions spanning multiple processes are future work.

4 Accelerating program idioms with TxFS

We now explore a number of programming idioms where a transactional API can improve performance because transactions provide the file system a sequence of operations which can be optimized as a group (§2). Whole transaction optimization can result in dramatic performance gains because the file system can eliminate temporary durable writes (such as the creation, use and deletion of a log file). In some cases, we show that benefits previously obtained by new interfaces (such as `osync` [2]) can be obtained easily with transactions.

4.1 Eliminating file creation

When an application creates a temporary file, syncs it, uses it, and then unlinks it (*e.g.,* logging shown in Figure 1b), enclosing the entire sequence in a transaction allows the file system to optimize out the file creation and all writes while maintaining crash consistency.

Workload	FS	TX
Create/unlink/sync	37.35s	0.28s (133×)
Logging	5.09s	4.23s (1.20×)
Ordering work	2.86 it/s	3.96 it/s (1.38×)

Table 1: Programming idioms sped up by TxFS transactions. Performance is measured in seconds (s), and iterations per second (it/s). Speedups for the transaction case are reported in parentheses.

The create/unlink/sync workload spawns six threads (one per core) where each thread repeatedly creates a file, unlinks it, and syncs the parent directory. Table 1 shows that placing the operation within a transaction increases performance by 133× because the transaction completely eliminates the workload's IO. While this test is an extreme case, we next look at using transactions to automatically convert a logging protocol into a more efficient update protocol.

4.2 Eliminating logging IO

Figure 1b shows the logging idiom used by modern applications to achieve crash consistency. Enclosing the entire protocol within a transaction allows the file system to transparently optimize this protocol into a more efficient direct modification. During a TxFS transaction, all sync-family calls are functional nops. Because the log file is created and deleted within the transaction, it does not need to be made persistent on transaction commit. Eliminating the persistence of the log file greatly reduces the amount of user data but also file system metadata (*e.g.,* block and inode bitmaps) that must be persisted.

Table 1 shows execution time for a microbenchmark that writes and syncs a log, and a version that encloses the entire protocol in a single TxFS transaction. Enclosing the logging protocol within a transaction increases performance by 20% and cuts the amount of IO performed in half because the log file is never persisted. Rewriting the code increases performance by 55% (3.28s, not shown in the table). In this case getting the most performance out of transactions requires rewriting the code to eliminate work that transactions make redundant. But even without a programmer rewrite, by just adding two lines of code to wrap a protocol in a transaction achieves 47% of the performance of doing a complete rewrite.

Optimizing SQLite logging with TxFS. Table 3 reports results for SQLite. "Rollback with TxFS" represents SQLite's default logging mode encased within a TxFS transaction. Just enclosing the logging activity with a transaction increases performance for updates by 14%. Modifying the code to eliminate the logging work that transactions make redundant increases the performance for updates to 31%, in part by reducing the number of

Experiment	TxFS benefit	Speed
Single-threaded SQLite	Faster IO path, Less sync	1.31×
TPC-C	Faster IO path, Less sync	1.61×
Android Mail	Cross abstraction	2.31×
Git	Crash consistency	1×

Table 2: The table summarizes the micro- and macro-benchmarks used to evaluate TxFS, and the speedup obtained in each experiment.

system calls by 2.5×.

4.3 Separating ordering and durability

Table 1 shows throughput for a workload that creates three 10MB files and then updates 10MB of a separate 40MB file. The user would like to create the files first, then update the data file. This type of ordering constraint often occurs in systems like Git that create log files and other files that hold intermediate state.

The first version uses `fsync()` to order the operations, while the second uses transactions that allow the first three file create operations to execute in any order, but they are all serialized behind the final data update transaction (using flags to `fs_tx_begin()` and `fs_tx_commit()`). The transactional approach has 38% higher throughput because the ordering constraints are decoupled from the persistence constraints. The work that first distinguished ordering from persistence suggests adding different flavor sync system calls [2], but TxFS can achieve the same result with transactions.

5 Evaluation

We evaluate the performance and durability guarantees of TxFS on a variety of micro-benchmarks and real workloads. The micro-benchmarks help point out how TxFS achieves specific design goals while the larger benchmarks validate that transactions provide stronger crash semantics and improved performance to a variety of large applications with minimal porting effort.

Testbed. Our experimental testbed consists of a machine with a 4 core Intel Xeon E3-1220 CPU and 32 GB DDR3 RAM and a machine with a 6 core Intel Xeon E5-2620 CPU and 8 GB DDR3 RAM. All experiments are performed on Ubuntu 16.04 LTS (Linux kernel 3.18.22). The kernel is installed on a Samsung 850 (512 GB) SSD and all experiments are done on a Samsung 850 (250 GB) SSD. The experimental SSD is run at low utilization (around 20%) to prevent confounding factors from wear leveling firmware.

Table 2 presents a summary of the different experiments used to evaluate TxFS and the speedup obtained in each experiment. In the Git experiment, TxFS provides strong crash-consistency guarantees without degrading performance. Note that if not explicitly mentioned, all our baselines run on ext4 with its default journaling mode, the ordered journaling mode.

5.1 Crash consistency

TxFS's ACID transactions should be recoverable after a system crash. In order to verify this crucial correctness property, we boot a virtual machine and run a script that creates many types of transactions in multiple threads with random amounts of contained work and conflict probabilities. We crash the VM at a random time and make sure the file system journal is recoverable and that the file system passes all fsck checks. We have run over 100 random crashes and can recover the file system in all cases. An alternate way to test crash consistency would use a testing framework such as CrashMonkey [13].

5.2 Stress testing TxFS

We performed stress testing on TxFS to ensure its correctness in the face of conflicts and multi-threaded operations. Our stress tests had two main workloads. Our first workload was a micro-benchmark with six threads starting TxFS transactions and performing file-system operations picked at random across two files before committing. These threads generate a lot of conflicts, stressing TxFS conflict detection and isolation mechanisms. Our second workload uses the SQLite embedded database, performing a number of database operations with multiple threads. We were able to run both workloads for over 24 hours on TxFS without a kernel crash or our unit tests failing, giving us a measure of confidence in the correctness and stability of the codebase.

5.3 SQLite

We modified SQLite to use TxFS transactions. Data and metadata are first written safely to the journal and then checkpointed in-place into the file system. Note that all metadata is written into the file system exactly once. With SQLite in write-ahead-logging (WAL) mode, metadata is written twice: once to SQLite's log and once to the actual database file. The size and frequency of metadata updates for SQLite is significant because in order to be recoverable, it must update the parent directory whenever log files are created or deleted [29]. We use `PRAGMA synchronous=NORMAL` (default) for all modes, and `PRAGMA wal_checkpoint(FULL)` for WAL mode to guarantee all ACID properties.

When SQLite uses TxFS transactions, crashes do not leave any residual files on storage. Currently, users often must remove these residual files by hand which is tedious and error-prone. TxFS transactions eliminate user-visible log files; user-level code sees only the before and after state of the database, not messy in-flight data.

Single-threaded SQLite. Table 3 shows that TxFS is the best performing option for SQLite updates. Data is

	Performance (kOps/s)		IO (GB)		Sync/tx	
Journal mode	Insert	Update	Insert	Update	Insert	Update
Rollback (default)	53.9	28.0	1.9	3.9	4	10
Truncate	53.5 (0.99×)	28.9 (1.03×)	1.9	3.9	4	10
WAL	39.8 (0.74×)	34.6 (1.23×)	3.9	3.8	3	3
TxFS	51.4 (0.95×)	36.7 (1.31×)	1.9	3.8	1	1
Rollback with TxFS	52.1 (0.97×)	31.9 (1.14×)	1.9	3.8	1	1
No journal (**unsafe**)	54.9 (1.02×)	50.6 (1.81×)	1.9	1.9	1	1

Table 3: The table compares operations per second (larger is better) and total amount of IO for SQLite executing 1.5M 1KB operations grouping 10K operations in a transaction using different journaling modes (including TxFS). The database is pre-populated with 15M rows. All experiments use SQLite's synchronous mode (its default).

	Rollback (default)	Truncate	WAL	TxFS	No journal (**unsafe**)
Delivery	110.52	123.33	157.01	188	300.4
New Order	142.38	165.15	216.8	240.34	445.14
Order Status	1998.53	2067.29	3317.1	2489.94	3141.13
Payment	198.45	240.21	367.26	300.61	909.91
Stock levl	575.03	602.33	765.41	684.06	1079.85
Total	172.97	203.3 (1.18×)	280.01 (1.62×)	278.97 (1.61×)	600.15 (3.47×)
Syscall/tx	208.0	207.95	138.26	100.35	146.9
Sync/tx	2.76	2.75	2.76	0.92	0.92
R MB/tx	0.018	0.016	0.013	0.013	0.007
W MB/tx	0.17	0.158	0.131	0.129	0.066
T MB/tx	0.187	0.174 (0.93×)	0.144 (0.77×)	0.142 (0.76×)	0.073 (0.39×)

Table 4: Rates (in transactions per second) for the TPC-C workload using different SQLite journaling modes. Each workload runs continuously for a fixed amount of time.

the average of five trials with standard deviations below 2.2% of the mean. For the update workload, TxFS is 31% faster than the default. We report IO totals as part of our validation that TxFS correctly writes all data in a crash-consistent manner. Several choices for SQLite logging mode, including TxFS, result in similar levels of IO that resemble the no-journal lower bound. Write-ahead logging mode (WAL) writes more data for the insert workload, which harms its performance. Note that TxFS does not suffer WAL's performance shortfall on insert, and TxFS surpasses WAL's performance on update, making it a better alternative. Although the file system journal shares similarity with a WAL log, TxFS does not generate redundant IO on insert because of its selective data journaling.

We run similar experiments with small updates (16 bytes) and find that there is little difference in performance between SQLite's different modes and TxFS. This shows that small transactions do not have significant overhead in TxFS.

TxFS's improves performance for the update workload is due to several factors. TxFS reduces the number of data syncs from 10 (in Rollback and Truncate mode) or 3 (in WAL mode) to only 1, which leads to better batching and re-ordering of writes inside a single transaction. It performs half of its IO to the journal, which is written sequentially. The remaining IO is done asynchronously via a periodic file-system checkpoint that writes the journaled blocks to in-place files. Since TxFS uses the file-system journal instead of an application-level journal for logging the transaction, it avoids the *journaling on journal* problem [25], where the journaling of the application-level log causes a significant slowdown. Even in realistic settings where performance is at a premium, transactions provide a simple, clean interface to get significantly increased file-system performance, while maintaining crash safety.

5.4 TPC-C

We run a version of the TPC-C benchmark [17], ported to use single-threaded SQLite[1]. TPC-C is a standard online transaction processing benchmark for an order-entry environment. R MB/tx is the amount of read IO per transaction, W is written IO and T is total.

Table 4 shows that TxFS outperforms SQLite's default mode by 1.61×. The performance advantage comes from two sources. First, TxFS writes less data and batches its writes. TxFS writes much of its data sequentially to the file system journal on `fs_tx_commit()` and writes back the journal data asynchronously. SQLite's default mode must write data to the SQLite journal and to the database file on `fsync()`. Therefore, TxFS writes only once in the critical path (to the journal), while SQLite (as configured in Section 5.3) must write to the journal plus database in the critical path. Second, TxFS decreases the number of system calls, especially sync-family calls. Table 4 shows that TxFS reduces the number of sync-family calls per transaction by 3×. By reducing the sync-familly calls, TxFS can batch writes in a transaction, reducing the amount of writes by 31.7% compared to default mode.

The performance of TxFS and WAL is similar. When transactions contain writes, TxFS has better performance than WAL, but it has worse performance for read-only

[1] `https://github.com/apavlo/py-tpcc/wiki/SQLite-Driver`

Journal mode	Throughput	IO(MB)
Rollback (default)	45.73	3269
Truncate	45.48 (0.99×)	3154
WAL	53.43 (1.17×)	3539
TxFS	105.68(2.31×)	6797
TxFS Small tx	60.85 (1.33×)	4052
No journal (**unsafe**)	61.88 (1.35×)	3995

Table 5: TxFS supports transactions across storage abstractions. Performance is measured in iterations per second.

transactions: WAL is 28% faster than TxFS for read-only transactions. “Order status” and “Stock level” consist of 3 select queries and 2 select queries respectively, resulting in lower throughput for TxFS compared with WAL. However, “Delivery” consists of 3 select, 3 update, and 1 delete queries, so TxFS outperforms WAL by 20%.

5.5 Abstractions built on files

Modern file systems support storage of not only files but databases (*e.g.,* SQLite) and key-value stores (*e.g.,* LevelDB and RocksDB). These abstractions are built on the file system and generally are a lower-performing, but easier to set up and maintain alternative to their dedicated counterparts.

TxFS supports transactions that span storage abstractions. Table 5 shows the throughput for a workload that models the core activity of Android mail, storing an image file and recording the path to that file in a SQLite database along with other metadata. The database is prepopulated with 100,000 1KB rows, image files are 1 MB. The workload creates the database record in one transaction, creates a uniquely named file where it stores the file data, syncs the data, and then updates the database record in a second transaction.

TxFS outperforms default SQLite by 2.31× and the best alternative (WAL mode) by 1.98×. It is essential to TxFS’s performance that both database transactions as well as the file system operation are all contained in a single transaction. When they are separate transactions (TxFS Small tx), performance is bounded by SQLite (i.e., it is close to no journaling). IO is not a bottleneck for this workload. The amount of IO performed is proportional to the amount of work done: TxFS has higher throughput, so it performs more IO.

5.6 Git

Git is a widely-used version control system. Git commands such as `git add` and `git commit` result in a large number of file-system operations. Git updates files by creating a temporary file, writing the desired data to it, and renaming it over the old file. To enable high performance, Git does not order its operations via `fsync()` [21], leaving it vulnerable to garbage files and outright data corruption on a system crash.

Category	System	Isolation	Durability	Easy-to-use API	Hardware independence	Performance	Complexity
In-kernel transactional FS	**TXFS**	✓	✓	✓	✓	H	L
	Valor	✓	✓	✗	✓	H	L
	TxF	✓	✓	✗	✓	H	H
Transactional OS	TxOS	✓	✓	✓	✓	H	H
FS over userspace databases	OdeFS Inversion DBFS Amino	Relying on databases		✗	✓	L	L
Transactional storage	CFS	✗	✓	✓	✗	H	L
	MARS	✓	✓	✗	✗	H	H
	Isotope	✓	✓	✓	✓	H	H
Failure atomicity	msync	✗	✓	✓	✓	H	L
	AdvFS	✗	✓	✓	✓	H	L

Table 6: The table compares prior work providing ACID transactions or failure atomicity in a local file system. Legend: ✓- supported, ✗- unsupported, L - Low, H - High. Note that only TxFS provides isolation and durability with high performance and low implementation complexity without restrictions or hardware modifications.

In our experiment, we run Git inside a virtual machine. We instrument the Git code to crash the VM at vulnerable points (such as after the temp file rename, but before the file is persistent). The workload first initializes a Git repository, populates it with 20,000 empty files, then adds all files at once.

After a VM restart, we find that the `.git/index` file has been truncated to zero bytes, resulting in a loss of the working tree. Running the Git recovery command `git fsck` simply reports a fatal error. Recovery is not possible unless the data has been backed up in another location. In contrast, when we change Git to use TxFS transactions, we find that crashes no longer produce such catastrophic errors. Furthermore, we do not find a significant difference in performance between the code that use TxFS transactions, and the code that does not. Thus, using TxFS transactions provides crash consistency for Git without any performance overhead.

6 Related work

There have been a number of efforts over the years to provide systems support for file-system transactions. Each of these systems failed to gain adoption due to one of the following reasons: they had severe restrictions on what could be placed inside a transaction, they were complicated to use, they added complexity to the kernel, or they caused significant performance degradation. Learning from prior systems, TxFS avoids all of these mistakes. Table 6 summarizes related work and demonstrates that TxFS is unique among transactional file systems.

Building file systems on top of user-space databases. One way to provide transactional updates for applications is to build a file system over a user-space transactional database. OdeFS [5], Inversion [18], and DBFS [16] use a database (such as Berkeley DB [19]) to provide ACID transactions to applications via NFS. Amino [31] tracks all user updates via `ptrace` and employs a user-level database to provide transactional updates. Such systems come with significant performance cost (*e.g.,* 50-80% for large operations in DBFS [16]).

In-kernel transactional file systems. An approach that leads to higher performance is adding transactions to in-kernel file systems. Valor [28] provides kernel support for file-system transactions. However, Valor does not provide a simple begin/end transaction interface, and it forces programmers to use seven new system calls to manage the transaction log.

Microsoft introduced Transactional NTFS (TxF), Transaction Registry (TxR), and the kernel transaction manager (KTM) in Windows Vista [24]. Using TxF requires all transactional operations be explicit (i.e., instead of using `read()` in a transaction, the programmer must add an explicit transactional read). Therefore TxF had a high barrier to entry and code that used it required separate maintenance. TxF also had significant limitations, like no transactions on the root file system.

Transactional operating systems. A third, somewhat heavyweight, approach is modifying the entire operating system to provide transactions. Our prior work, TxOS [22], is an operating system that provides transactions. This approach adds significant complexity to the kernel. For example, TxOS modified tens of thousands of lines of code and changed core OS data structures like the inode. Maintaining such a kernel will be tricky – Windows abandoned its transactional file system and kernel transaction manager [14].

The transactional capabilities of the file system supported by TxOS is similar in approach to TxFS. It also uses the file-system journal and modifies the virtual file system (VFS) code to provide isolation. One could view TxFS as specializing TxOS to the file system, achieving a transactional file system at significantly lower cost, while adding file-system specific optimizations like selective journaling and eliminating redundant work within transactions.

Transactional storage systems. Similar to our work, CFS [15] provides a lightweight mechanism for atomic updates of multiple files, building on top of transactional flash storage. MARS [3] builds on hardware-provided atomicity to build a transactional system. TxFlash [23] uses the copy-on-write nature of Flash SSDs to provide transactions at low cost. In contrast to these systems, TxFS provides transactions without assuming any hardware support (beside device cache flush and atomic sector updates). Isotope [26] uses multi-version concurrency control to provide isolation, significantly increasing its complexity. Isotope builds a user-space transactional file system using FUSE, which limits its performance for certain workloads. The higher abstraction level of TxFS makes implementing transactional optimizations and tailored isolation significantly easier than the lower level of Isotope.

Failure atomicity. Failure-atomic msync [20] is similar to TxFS in that it re-uses the journal for providing atomicity to application updates; in contrast, TxFS provides full ACID transactions at significantly higher complexity. AdvFS [30] is also limited in the same way, is specific to the Tru64 file system, and is not available as open-source (latest version available was from 2008). The principles behind TxFS could be used in any file system that has an internal mechanism for atomic updates.

We previewed the ideas behind TxFS at HotOS [11], but this paper reports on the completed system with comprehensive evaluation.

7 Conclusion

We present TxFS, a transactional file system built with less development effort than previous systems by leveraging the file-system journal. TxFS is easy to develop, it is easy to use, and it does not have significant overhead for transactions. We show that using TxFS transactions increases performance significantly for a number of different workloads.

Transactional file systems have not been successful for a variety of reasons. TxFS shows that it is possible to avoid the mistakes of the past, and build a transactional file system with low complexity. Given the power and flexibility of file-system transactions, we believe they should be examined again by file-system researchers and developers. Adopting a transactional interface would allow us to borrow decades of research on optimizations from the database community while greatly simplifying the development of crash-consistent applications.

Acknowledgement

We would like to thank the anonymous reviewers for their feedback and guidance. This work is supported in part by NSF (CNS-1618563, CCF-1333594) and by generous donations from VMware, Facebook, and Google.

References

[1] Fsync man page. http://man7.org/linux/man-pages/man2/fdatasync.2.html.

[2] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[3] Coburn, Joel and Bunker, Trevor and Schwarz, Meir and Gupta, Rajesh and Swanson, Steven. From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, New York, NY, USA, 2013. ACM.

[4] David A. Wheeler. Sloccount. https://www.dwheeler.com/sloccount/.

[5] Gehani, Narain H and Jagadish, HV and Roome, William D. OdeFS: A File System Interface to an Object-Oriented Database. In *VLDB*, pages 249–260. Citeseer, 1994.

[6] Jim Gray. The Transaction Concept: Virtues and Limitations. In *VLDB*, volume 81, pages 144–154, 1981.

[7] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In G. M. Nijssen, editor, *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, pages 365–394. North-Holland, 1976.

[8] Robert Hagmann. *Reimplementing the Cedar File System Using Logging and Group Commit*, volume 21. ACM, 1987.

[9] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. *ACM Transactions on Computer Systems (TOCS)*, 30(3):10, 2012.

[10] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[11] Yige Hu, Younjin Kwon, Vijay Chidambaram, and Emmett Witchel. From Crash Consistency to Transactions. In *16th Workshop on Hot Topics in Operating Systems (HotOS 17)*, Whistler, Canada, 2017.

[12] Sitaram Iyer and Peter Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, 2001.

[13] Ashlie Martinez and Vijay Chidambaram. CrashMonkey: A Framework to Systematically Test File-System Crash Consistency. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[14] Microsoft. Alternatives to using transactional ntfs. "https://msdn.microsoft.com/en-us/en-%20us/library/hh802690.aspx".

[15] Min, Changwoo and Kang, Woon-Hak and Kim, Taesoo and Lee, Sang-Won and Eom, Young Ik. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 221–234, 2015.

[16] Nick Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system. https://goo.gl/3Gj328, 2002.

[17] Raghunath Nambiar, Meikel Poess, Andrew Masland, H. Reza Taheri, Andrew Bond, Forrest Carman, and Michael Majdalany. TPC state of the council 2013. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*, volume 8391 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2013.

[18] Michael A Olson. The Design and Implementation of the Inversion File System. In *USENIX Winter*, pages 205–218, 1993.

[19] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[20] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 225–238. ACM, 2013.

[21] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[22] Donald E Porter, Owen S Hofmann, Christopher J Rossbach, Alexander Benn, and Emmett Witchel. Operating System Transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 161–176. ACM, 2009.

[23] Prabhakaran, Vijayan and Rodeheffer, Thomas L and Zhou, Lidong. Transactional Flash. In *OSDI*, pages 147–160, 2008.

[24] Russinovich, Mark E and Solomon, David A and Allchin, Jim. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*, volume 4. Microsoft Press Redmond, 2005.

[25] Kai Shen, Stan Park, and Men Zhu. Journaling of Journal is (Almost) Free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 287–293, 2014.

[26] Shin, Ji-Yong and Balakrishnan, Mahesh and Marian, Tudor and Weatherspoon, Hakim. Isotope: Transactional Isolation for Block Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[27] Riley Spahn, Jonathan Bell, Michael Lee, Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 113–129, 2014.

[28] Richard P Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P Wright. Enabling Transactional File Access via Lightweight Kernel Extensions. In *FAST*, volume 9, pages 29–42, 2009.

[29] SQLite. SQLite transactional SQL database engine. `http://www.sqlite.org/`.

[30] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya S Mannarswamy, Terence Kelly, and Charles B Morrey III. Failure-Atomic Updates of Application Data in a Linux File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 203–211, 2015.

[31] Wright, Charles P and Spillane, Richard and Sivathanu, Gopalan and Zadok, Erez. Extending ACID Semantics to the File System. *ACM Transactions on Storage (TOS)*, 3(2):4, 2007.

Towards Better Understanding of Black-box Auto-Tuning: A Comparative Analysis for Storage Systems

Zhen Cao[1], Vasily Tarasov[2], Sachin Tiwari[1], and Erez Zadok[1]
[1]*Stony Brook University* *and* [2]*IBM Research—Almaden*

Abstract

Modern computer systems come with a large number of configurable parameters that control their behavior. Tuning system parameters can provide significant gains in performance but is challenging because of the immense number of configurations and complex, nonlinear system behavior. In recent years, several studies attempted to automate the tuning of system configurations; but they all applied only one or few optimization methods. In this paper, for the first time, we apply and then perform comparative analysis of multiple black-box optimization techniques on storage systems, which are often the slowest components of computing systems. Our experiments were conducted on a parameter space consisting of nearly 25,000 unique configurations and over 450,000 data points. We compared these methods for their ability to find near-optimal configurations, convergence time, and instantaneous system throughput during auto-tuning. We found that optimal configurations differed by hardware, software, and workloads—and that no one technique was superior to all others. Based on the results and domain expertise, we begin to explain the efficacy of these important automated black-box optimization methods from a systems perspective.

1 Introduction

Storage is a critical element of computer systems and key to data-intensive applications. Storage systems come with a vast number of configurable parameters that control system's behavior. Ext4 alone has around 60 parameters with whopping 10^{37} unique combinations of values. Default parameter settings provided by vendors are often suboptimal for a specific user deployment; previous research showed that tuning even a small subset of parameters can improve power and performance efficiency of storage systems by as much as 9× [66].

Traditionally, system administrators pick parameter settings based on their expertise and experience. Due to the increased complexity of storage systems, however, manual tuning does not scale well [87]. Recently, several attempts were made to automate the tuning of computer systems in general and storage systems in particular [71, 78]. Black-box auto-tuning is an especially popular approach thanks to its obliviousness to a system's internals [86]. For example, Genetic Algorithms (GA) were applied to optimize the I/O performance of HDF5-based applications [5] and Bayesian Optimization (BO) was used to find a near-optimal configuration for Cloud VMs [3]. Other methods include Evolutionary Strategies [62], Smart Hill-Climbing [84], and Simulated Annealing [21]. The basic mechanism behind black-box auto-tuning is to iteratively try different configurations, measure an objective function's value—and based on the previously learned information—select the next configurations to try. For storage systems, objective functions can be throughput, energy consumption, purchase cost, or even a formula combining different metrics [50, 71]. Despite some appealing results, there is no deep understanding how exactly these methods work, their efficacy and efficiency, and which methods are more suitable for which problems. Moreover, previous works evaluated only one or few algorithms at a time. In this paper, for the first time (to the best of our knowledge), we apply and analytically compare *multiple* black-box optimization techniques on storage systems.

To demonstrate and compare these algorithms' ability to find (near-)optimal configurations, we started by exhaustively evaluating several storage systems under four workloads on two servers with different hardware and storage devices; the largest system consisted of 6,222 unique configurations. Over a period of 2+ years, we executed 450,000+ experimental runs. We stored all data points in a relational database for query convenience, including hardware and workload details, throughput, energy consumption, running time, etc. In this paper, we focused on optimizing for throughput, but our methodology and observations are applicable to other metrics as well. We will release our dataset publicly to facilitate more research into auto-tuning and better understanding of storage systems.

Next, we applied several popular techniques to the collected dataset to find optimal configurations under various hardware and workload settings: Simulated Annealing (SA), Genetic Algorithms (GA), Bayesian Optimization (BO), and Deep Q-Networks (DQN). We also tried Random Search (RS) in our experiments, which showed surprisingly good results in previous research [8]. We compared these techniques from various aspects, such as the ability to find near-optimal configurations, convergence time, and instantaneous system throughput during auto-tuning. For example, we found that several techniques were able to converge to good configurations given enough time, but their efficacy differed a lot. GA and BO outperformed SA and

DQN on our parameter spaces, both in terms of convergence time and instantaneous throughputs. We also showed that hyper-parameter settings of these optimization algorithms, such as mutation rate in GA, could affect the tuning results. We further compared the techniques across three behavioral dimensions: (1) *Exploration*: how much the technique searches the space randomly. (2) *Exploitation*: how much the technique leverages the "neighborhood" of the current candidate or previous search history to find even better configurations. (3) *History*: how much data from previous evaluations is kept and utilized in the overall search process. We show that all techniques employ these three key concepts to varying degrees and the trade-off among them plays an important role in the effectiveness and efficiency of the algorithms. Based on our experimental results and domain expertise, we provide explanations of efficacy of such black-box optimization methods from a storage perspective. We observed that certain parameters would have a greater effect on system performance than others, and the set of dominant parameters depends on file systems and workloads. This allows us to provide more insights into the auto-tuning process.

Auto-tuning storage systems is fairly complex and challenging. We made several necessary assumptions and simplifications while collecting our exhaustive data, which we detail in §3. Therefore, some of our observations might differ when applied to production systems. However, the main purpose of this paper is *not* to provide a complete solution; rather, we focus on comparing and understanding the efficacy of several popular optimization techniques when applied to storage systems. We believe this paves the way for practical auto-tuning storage systems in real-time.

The rest of the paper is organized as follows. §2 explains the challenges of auto-tuning storage systems and provides necessary background knowledge. §3 describes our experimental methodology and environments. In §4 we applied multiple optimization methods and evaluated and explained them from various aspects. §5 covers limitations and future plans for our work. §6 lists related work. We conclude and discuss future directions in §7.

2 Background

Storage systems are often a critical component of computer systems, and are the foundation for many data-intensive applications. Usually they come with a large number of configurable options that could affect or even determine the systems' performance [12, 74], energy consumption [66], and other aspects [47, 71]. Here we define a *parameter* as one configurable option, and a *configuration* as a combination of parameter values. For example, the parameter *block_size* of Ext4 can take 3 values: 1K, 2K, and 4K. Based on this, *[journal_mode="data=writeback", block_size=4K, inode_size=4K]* is one *configuration* with 3 specific parameters: *journal mode*, *block size*, and *inode size*. All possible configurations form a *parameter space*.

When configuring storage systems, users often stick with the default configurations provided by vendors because 1) it is nearly impossible to know the impact of every parameter across multiple layers; and 2) vendors' default configurations are trusted to be "good enough". However, previous studies [66] showed that tuning even a tiny subset of parameters could improve the performance and energy efficiency for storage systems by as much as 9×. As technological progress slows down, it becomes even more important to squeeze every bit of performance out of deployed storage systems.

In the rest of this section we first discuss the challenges of system tuning (§2.1). Then, §2.2 briefly introduces several promising techniques that we explore in this paper. §2.3 explains certain methods that we deem less promising. §2.4 provides a unified view of these optimization methods.

2.1 Challenges

The tuning task for storage systems is difficult, due to the following four challenges.

(1) Large parameter space. Modern storage systems are fairly complex and easily come with hundreds or even thousands of tunable parameters. One evaluation for storage systems can take multiple minutes or even hours, which makes exhaustive search impractical. Even human experts cannot know the exact impact of every parameter and thus have little insight into how to optimize. For example, Ext4+NFS would result in a parameter space consisting of more than 10^{22} unique configurations. IBM's General Parallel File System (GPFS) [64] contains more than 100 tunable parameters, and hence 10^{40} configurations. From the hardware perspective, SSDs [30, 53, 57, 65], shingled drives [1, 2, 32, 45], and non-volatile memory [40, 83] are gaining popularity, plus more layers (LVM, RAID) are added.

(2) Non-linearity. A system is *non-linear* when the output is not directly proportional to the input. Many computer systems are non-linear [16], including storage systems [74]. For example, Figure 1 shows the average operation latency of GPFS under a typical database server workload while changing only the value of the parameter *pagepool* from 32MB to 128MB, and setting all the others to their default. Clearly the average latency is not directly proportional to the *pagepool* size. In fact, through our experiments, we have seen many more parameters with similar behavior. Worse, the parameter space for storage systems is often sparse, irregular, and contains multiple peaks. This makes automatic optimization even more challenging, as it has to avoid

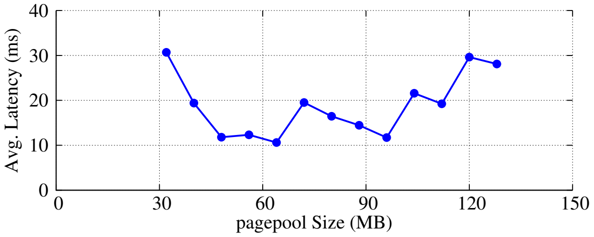


Figure 1: Storage systems are non-linear.

getting stuck in a local optima [36].

(3) Non-reusable results. Previous studies have shown that evaluation results of storage systems [12,66] and databases [78] are dependent on the specific hardware and workloads. One good configuration might perform poorly when the environment changes. Our evaluation results in Section 4 show similar observations.

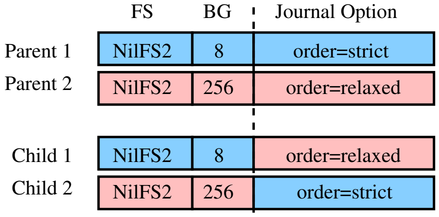
(4) Discrete and non-numeric parameters. Some storage system parameters can take continuous real values, while many others are discrete and take only a limited set of values. Some parameters are not numeric (e.g., I/O scheduler name or file system type). This adds difficulty in applying gradient-based approaches.

Given these challenges, manual tuning of storage systems becomes nearly impossible while automatic tuning merely difficult. In this paper we focus on automatic tuning and treat it as an optimization problem.

2.2 Applied Methods

Several classes of algorithms have been proposed for similar optimization tasks, including automated tuning for hyper-parameters of machine learning systems [7, 8, 59] and optimization of physical systems [3, 78]. Examples include Genetic Algorithms (GA) [18, 34], Simulated Annealing (SA) [15, 41], Bayesian Optimization (BO) [11,68], and Deep Q-Networks (DQN) [46,54,55]. Although these methods were proposed originally in different scholarly fields, they can all be characterized as black-box optimizations. In this section we introduce several of these techniques that we successfully applied in auto-tuning storage systems.

Simulated Annealing (SA) is inspired by the annealing process in metallurgy, which involves the heating and controlled cooling of a material to get to a state with minimum thermodynamic free energy. When applied to storage systems, a *state* corresponds to one *configuration*. *Neighbors* of a state refer to new configurations achieved by altering only one parameter value of the current state. The thermodynamic free energy is analogous to optimization objectives. SA works by maintaining the *temperature* of the system, which determines the probability of accepting a certain move. Instead of always moving towards better states as hill-climbing methods do, SA defines an *acceptance probability distribution*, which allows it to accept some bad moves in the short run, that can lead to even-better moves later on. The system is initialized with a high temperature, and thus has high probability of accepting worse states in the beginning. The temperature is gradually reduced based on a pre-defined *cooling schedule*, thus reducing the probability of accepting bad states over time.



	FS	BG	Journal Option
Parent 1	NilFS2	8	order=strict
Parent 2	NilFS2	256	order=relaxed
Child 1	NilFS2	8	order=relaxed
Child 2	NilFS2	256	order=strict

Figure 2: Crossover in Genetic Algorithm (GA).

Genetic Algorithms (GA) were inspired by the process of natural selection [34]. It maintains a population of *chromosomes* (configurations) and applies several genetic operators to them. *Crossover* takes two parent chromosomes and generates new ones. As Figure 2 illustrates, two parent Nilfs2 configurations are cut at the same *crossover* point, and then the subparts after the crossover point are exchanged between them to generate two new child configurations. Better chromosomes will have a higher probability to "survive" in future *selection* phases. *Mutation* randomly picks a chromosome and mutates one or more parameter values, which produces a completely different chromosome.

Reinforcement Learning (RL) [72] is an area of machine learning inspired by behaviorist psychology. RL explores how software agents take actions in an environment to maximize the defined cumulative rewards. Most RL algorithms can be formulated as a model consisting of: (1) A set of environment states; (2) A set of agent actions; and (3) A set of scalar rewards. In case of storage systems, *states* correspond to *configurations*, *actions* mean changing to a different configuration, and *rewards* are differences in evaluation results. The agent records its previous experience (history), and makes it available through a *value function*, which can be used to predict the expected reward of state-action pairs. The *policy* determines how the agent takes action, which maintains the *exploration-exploitation* trade-off. The value function can take a tabular form, but this does not scale well to many dimensions. Function approximation is proposed to deal with high dimensionality, which is still known to be unstable or even divergent. With recent advances in Deep Learning [28], deep convolutional neural networks, termed Deep Q-Networks (DQN), were proposed to parameterize the value function, and have been successfully applied in solving various problems [54, 55]. Many variants of DQN have been proposed [46].

Bayesian Optimization (BO) [11, 68] is a popular framework to solve optimization problems. It models

Algorithm	Origin	Exploration	Exploitation	History
Simulated Annealing (SA)	Annealing technology in metallurgy	Allowing moving to worse neighbor states	Neighbor function	N/A
Genetic Algorithms (GA)	Natural evolution	Mutation	Crossover and selection	Current population
Deep Q-Networks (DQN)	Behaviorist psychology and neuroscience	Taking random actions	Taking actions based on action-reward function	Deep convolutional neural network
Bayesian Optimization (BO)	Statistics and experimental design	Selecting samples with high variances	Selecting samples with high mean values	Acquisition function & probabilistic model

Table 1: Comparison and summaries of optimization techniques.

the objective function as a stochastic process, with the argument corresponding to one storage configuration. In the beginning, a set of prior points (configurations) are given to get a fair estimate of the entire parameter space. BO works by computing the *confidence interval* of the objective function according to previous evaluation results, which is defined as the range of values that the evaluation result is most likely to fall into (e.g., with 95% probability). The next configuration is selected based on a pre-defined *acquisition function*. Both confidence intervals and the acquisition function are updated with each new evaluation. BO has been successfully applied in various areas, including hyper-parameter optimization [17] and system configuration optimization [3]. BO and its variants differ mainly in their form of probabilistic models and acquisition functions. In this paper we focus mainly on Gaussian priors and an Expected Improvement acquisition function [68].

Other promising techniques include Tabu Search [27], Particle Swarm Optimization [39], Ant Colony Optimization [20], Memetic Algorithms [52], etc. Due to space limits, we omit comparing all of them in this paper (part of our future work). In fact, as detailed in §2.4, most of these techniques actually share similar traits.

2.3 Other Methods

Although many optimization techniques have been proposed, we feel that not all of them make good choices for auto-tuning storage systems. For example, since many parameters of storage systems are non-numeric, most gradient-based methods (i.e., based on linear-regression) are less suitable to this task [29].

Control Theory (CT). CT was historically used to manage linear system parameters [19, 37, 44]. CT builds a controller for a system so its output follows a desired reference signal [33, 43]. However, CT has been shown to have the following three problems: 1) CT tends to be unstable in controlling non-linear systems [48, 49]. Although some variants were proposed, they do not scale well. 2) CT cannot handle non-numeric parameters; and 3) CT requires a lot of data during the learning phase, called *identification* to build a good controller.

Supervised Machine Learning (ML). Supervised ML has been successfully applied in various domains [9, 10, 56, 81]. However, the accuracy of ML models depends heavily on the quality and amount of training data [81], which is not available or impossible to collect for large parameter spaces such as ours.

Therefore, we feel that neither CT nor supervised ML, in their current state, are the first choice to *directly and efficiently* apply for auto-tuning storage systems. That said, they constantly evolve and new promising results appear in research literature [4, 67, 69, 86]; we plan to investigate them in the future.

2.4 Unified Framework

Most optimization techniques are known to follow the *exploration-exploitation* dilemma [23, 46, 68, 79]. Here we summarize the aforementioned methods by extending the unified framework with a third factor, the *history*. Our unified view thus defines three factors or dimensions: ■ **(1) Exploration** defines how the technique searches unvisited areas. This often includes a combination of pure random and also guided search based on *history*. ■ **(2) Exploitation** defines how the technique leverages *history* to find next sample. ■ **(3) History** defines how much data from previous evaluations is kept. History information can be used to help guide both future exploration and exploitation (e.g., avoiding less promising regions, or selecting regions that have never been explored before). Table 1 summarizes how the aforementioned techniques work by maintaining the balance among these three key factors. For example, GA keeps the evaluation results from the last generation, which corresponds to the concept of *history*. GA then *exploits* the stored information, applying selection and crossover to search nearby areas and pick the next generation. Occasionally, it also randomly mutates some chosen parameters, which is the idea of *exploration*. As shown in §4, the trade-off among exploration, exploitation, and history determines the effectiveness and efficiency of these optimization techniques.

3 Experimental Settings

We now describe details of the experimental environments, parameter spaces, and our implementations of optimization algorithms.

Param.	Abbr.	Values
File System	FS	Ext2, Ext3, Ext4, XFS, Btrfs, Nilfs2, Reiserfs
Block (Leaf) Size	BS	1K, 2K, 4K
Inode Size, Sector Size	IS	n/a, 128, 256, 512, 1024, 2048, 4096, 8192
Block Group	BG	n/a, 2, 4, 8, 16, 32, 64, 128, 256
Journal Option	JO	n/a, order=strict, order=relaxed, data=journal, data=ordered, data=writeback
Atime Option	AO	relatime, noatime
Special Option	SO	n/a, compress, nodatacow, nodatasum, notail
I/O Scheduler	I/O	noop, cfq, deadline

Table 2: Details of parameter spaces.

Hardware. We performed experiments on two sets of machines with different hardware categorized as low-end (M1) and mid-range (M2). We list the hardware details in Table 3. We also use Watts Up Pro ES power meters to measure the energy consumption [82].

Workload. We benchmarked storage configuration with four typical macro-workloads generated by Filebench [25, 75]. ■ **(1) Mailserver** emulates the I/O workload of a multi-threaded email server. ■ **(2) Fileserver** emulates the I/O workload of a server that hosts users' home directories. ■ **(3) Webserver** emulates the I/O workload of a typical static Web server with a high percentage of reads. ■ **(4) Dbserver** mimics the behaviors of Online Transaction Processing (OLTP) databases. Before each experiment run, we formatted and mounted the storage devices with the targeted file system.

The working set size affects the duration of an experiment [74]. Our goal in this study was to explore a large set of parameters and values quickly (although it still took us over two years). We therefore decided to trade the working set size in favor of increasing the number of configurations we could explore in a practical time period. We used the default working set sizes in Filebench, and ran each workload for 100 seconds; this is long enough to get stable evaluation results under this setting. The experiments demonstrate a wide range of performance numbers and are suitable for evaluating different optimization methods.

Parameter Space. Since the main goal of our paper is to compare multiple optimization techniques, we want our storage parameter spaces to be large and complex enough. Alas, evaluations for storage systems take a long time. Considering experimentation on multiple hardware settings and workloads, we decided to experiment with a reasonable subset of the most relevant storage system parameters. We selected parameters in close collaboration with several storage experts that have either contributed to storage stack designs or have spent years tuning storage systems in the field. We experimented with 7 Linux file systems that span a wide range of designs and features: *Ext2* [13], *Ext3* [77], *Ext4* [24], *XFS* [73], *Btrfs* [61], *Nilfs2* [42], and *Reiserfs* [60].

Hardware	M1	M2
Model	Dell PE SC1425	Dell PE R710
CPU	Intel Xeon single-core 2.8GHz CPU × 2	Intel Xeon quad-core 2.4GHz CPU × 2
Memory	2GB	24GB
Storage	HDD1 (73GB Seagate ST373207LW SCSI drive)	HDD2 (147GB SAS), HDD3 (500GB SAS), HDD4 (250GB SATA), SSD (200GB)

Table 3: Details of experiment machines.

Our experiments were mainly conducted on two sets of parameters, termed as *Storage V1* and *Storage V2*. We started with seven common file system parameters (shown in the first 7 rows of Table 2), and refer it as *Storage V1*. *Storage V1* was tested on M1 machines. We then extended our search space with one more parameter, the *I/O Scheduler*, and refer to it as *Storage V2*. *Storage V2* was evaluated on M2 servers. Note that certain combinations of parameter values could produce invalid configurations. For example, for Ext2, the journaling option makes no sense because Ext2 does not have a journal. To handle this, we added a value *n/a* to the existing range of parameters. Any parameter with *n/a* value is considered invalid. Invalid configurations will always come with evaluation results of zero (i.e., no throughput); this ensures they are purged in an upcoming optimization process. There are 2,074 valid configurations in *Storage V1* and 6,222 in *Storage V2* for each workload and storage device. We believe our search spaces are large and complex enough to demonstrate the difference in efficiency of various optimization algorithms. Furthermore, many of the chosen parameters are commonly tuned and studied by storage experts; having a basic understanding of such parameters helped us understand auto-tuning results.

Experiments and implementations. Our experiments and implementation consist of two parts. First, we exhaustively ran all configurations for each workload and device on M1 and M2 machines, and stored the results in a relational database. We collected the throughput in terms of I/O operations per second, as reported by Filebench, the running time (including setup time), as well as power and energy consumption. To acquire more accurate and stable results, we evaluated each configuration under the same environment for at least 3 runs, resulting in more than 450,000 total experimental runs. This data collection benefited our evaluation on auto-tuning as we can simply simulate a variety of algorithms by just querying the database for the evaluation results for different configurations, without having to rerun slow I/O experiments. The exhaustive search

Hardware-Workload-Device	File System	Block Size	Inode Size	BG Count	Journal Options	Atime Options	Special Options	I/O Scheduler	Through-put (IOPS)
M1-Mail-HDD1	Nilfs2	2K	n/a	256	order=relaxed	relatime	n/a	-	3,677
M2-Mail-HDD3	Ext2	4K	256	32	n/a	relatime	n/a	noop	18,744
M2-File-HDD3	Btrfs	4K	4,096	n/a	n/a	relatime	compress	deadline	16,549
M2-Mail-SSD	Ext2	4K	256	8	n/a	relatime	n/a	noop	18,845
M2-DB-SSD	Ext4	1K	128	2	data=ordered	noatime	n/a	noop	41,948
M2-Web-SSD	Ext4	4K	128	4	data=ordered	noatime	n/a	noop	16,185

Table 4: Global optimal configurations with different settings and workloads.

also lets us know exactly what the global optimal configurations are, so that we can calculate how close each optimization method gets to the global optimum.

Second, we simulated the process of auto-tuning storage systems by running the desired optimization method and querying the database for the average evaluation results from multiple (3+) repeated runs. We focused on optimizing for throughput in this paper. The computation cost of optimization algorithms are ignored in our experiments. We believe our observations are applicable to other optimization objectives as well. Our implementations of optimization methods are mostly based on open-source libraries. We use Pyevolve [58] for Genetic Algorithms, Scikit-Optimize [70] for Bayesian Optimization, and TensorFlow [76] for the DQN implementation. We implemented a simple version of Simulated Annealing, with both linear and geometric cooling schedules. (We also fixed bugs in Pyevolve and plan to release our patches.) Most of our implementation was done by converting storage-related concepts into algorithm-specific ones. For example, for GA, we defined each storage parameter as a *gene*, and each configuration as a *chromosome*. For DQN we provided storage-specific definitions for states, actions, and rewards. The complete implementation uses around 10,000 lines of Python code.

4 Evaluations

Our evaluation mainly focuses on comparing the effectiveness and speed of applying multiple optimization techniques on auto-tuning storage systems, and providing insights into our observations. §4.1 overviews the data sets that we collected for over two years. §4.2 compares five popular optimization techniques from several aspects. §4.3 uses GA as a case study to show that hyper-parameters of these methods can also impact the auto-tuning results. §4.4 takes the first step towards explaining these black-box optimization methods, based on our evaluation results and our storage expertise.

4.1 Overview of Data Sets

As per §3, our experimental methodology is to first exhaustively run all configurations under different workloads and test machines. We stored the results in a database for future use. This data collection benefits future experiments as we can simulate a variety of algorithms by querying the database for the evaluation results of different configurations. Due to space limits, in this section we show only 6 representative data sets out of 18: 2 workloads on M1 and 4 devices × 4 workloads on M2. They were picked to (1) show a wide range of hardware and workloads' impact on optimization results and (2) to present more SSD results, given SSDs' increasing popularity.

Figure 3 shows the throughput CDF among all configurations for each hardware setting and workload. The Y-axis is normalized by the maximum throughput under each experiment setting. The symbols on each line mark the default Ext4 configurations. As seen, for most settings, throughput values vary across a wide range. The ratios of the worst throughput to the best one are mostly between 0.2–0.4. In one extreme case, for *Fileserver* on *M2* machines and with the HDD3 device (abbreviated as *M2-Fileserver-HDD3*), the worst configuration only produces 1% I/O operations per unit time, compared with the global optimal one. This underlines the importance of tuning storage systems: an improperly configured system could be remarkably under-utilized, and thus wasting a lot of resources. However, *M2-Webserver-SSD* shows a much narrower range of throughput, with the worst-to-best ratio close to 0.9. This is attributed mainly to the fact that *Webserver* consists of mostly sequential read operations that are processed similarly by different I/O stack configurations. Figure 3 also shows that default Ext4 configurations are always sub-optimal and, under most settings, ranked lower than the top 40% configurations. For *M1-Mailserver-HDD1*, the default Ext4 configuration shows a normalized throughput of 0.39, which means that the optimal configuration performs 2.5 times better.

Table 4 lists optimal configurations for the same six hardware and workload settings. As we can see, optimal configurations depend on the specific hardware as well as the running workload. For *M1-Mailserver-HDD1*, the global best is a *Nilfs2* configuration. However, if we fix the workload, change the hardware, and get *M2-Mailserver-HDD3*, the optimum becomes an *Ext4* configuration. Similarly, fixing the hardware to *M2-*-SSD* and experimenting under different workloads leads to different optimal configurations. This proves our early

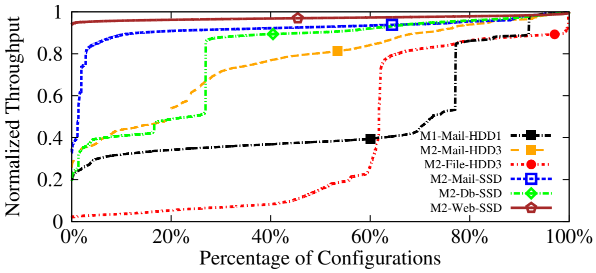


Figure 3: Throughput CDF with different hardware and workloads, with symbols marking the default Ext4 configurations.

claim that performance is sensitive to the environment (e.g., hardware, configuration, and workloads); this actually complicates the problem as results from one environment cannot be directly applied in another.

4.2 Comparative Analysis

Many optimization techniques have been applied to various auto-tuning tasks [71, 78]. However, previous efforts picked algorithms somewhat arbitrarily and evaluated only one algorithm at a time. Here we provide the first comparative study of multiple black-box optimization techniques on auto-tuning storage systems. As discussed in §2.2, we focus our evaluations on a representative set of optimization methods, and their common hyper-parameter settings, including 1) Simulated Annealing (SA), with a linear cooling schedule; 2) Genetic Algorithms (GAs) with population size of 8, mutation rate of 2%; 3) Deep Q-Networks (DQN) with experience replay [55] and $\epsilon = 0.2$, where ϵ; represents the probability of an agent taking random actions. 4) Bayesian Optimization (BO) with Expected Improvement (EI) and Gaussian prior; and 5) Random Search (RS), which merely performs random selection without replacement. We provide more discussion on the impact of hyper-parameters in §4.3. Note that SA, DQN, and RS experiments start with the default Ext4 configuration. GA and BO require several initial configurations (*prior points*), which we set to default configurations of all seven file systems. This allows us to simulate real-world use cases, where users often deploy their system with the default settings (and may manually optimize starting from the defaults). In the current experiments we assume that changing parameter values comes at no cost. In reality, parameters like *Block Size* may require re-formatting file systems.

Figure 4 presents one simulated run of each optimization method on *M2-Mailserver-HDD3*; the Y-axis shows the throughput value of the best configuration found so far, and the X-axis is the running time. All time-related metrics in this paper are based on the actual running time of evaluating each storage configuration, which is stored in our database. This includes both setup time and benchmarking time. We are not comparing the running

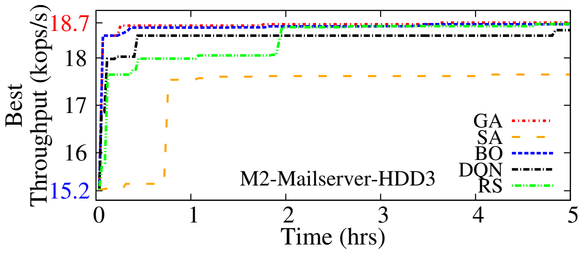


Figure 4: Highest throughput found over time, zooming in the $Y \in [15 : 19]$ range. The blue number (15.2) on the Y-axis shows the default, and the red one (18.7) shows the optimal.

costs (including any necessary training phases) for optimization methods here, which is our future work. Figure 4 is plotted by zooming in the range of $Y \in [15 : 19]$, with the blue number (15.2) on Y-axis represents the default, while the red one (18.7) shows the global optimal. Here we define a near-optimal configuration as one with throughput higher than 99% of the global optimal value. As shown in Figure 4, all five methods were able to gradually find higher performing configurations, but their effectiveness and speed differed a lot. SA performed the worst, and got stuck in a configuration with throughput value of less than 18K IOps. DQN was able to converge to a good configuration, but spent more time to achieve that than RS. GA and BO performed best out of these five tested optimization methods. They both successfully identified a near-optimal configuration within one hour. Interestingly, we observed that pure Random Search (RS) produced better results than some other optimization methods; the reason is that within the search space *M2-Mailserver-HDD3*, 4.5% of total configurations are near-optimal. RS needs only to hit one of them to reach good auto-tuning results.

Since exploration is one critical component of optimizations (see §2.4), their evaluation results could also exhibit some degree of randomness. To compare them more thoroughly, we ran each optimization technique on the same environment for 1,000 runs. Figure 5 shows the results, which evaluate the techniques' probability to find near-optimal configurations, defined the same as in Figure 4. The Y-axis shows the percentage of total runs that found a near-optimal configuration within a certain time (X-axis). Under *M2-Mailserver-HDD3*, seen in the upper part of Figure 5, SA had the lowest probability among 5 algorithms. Even after 5 hours, only around 80% of its runs found one near-optimal configuration, which shows that SA sometimes gets stuck in a local optimum. For other optimization methods, given enough time, over 90% of their runs converged to a near-optimal configuration, with BO outperforming GA, and GA outperforming DQN. RS shows the highest probability of finding near-optimal configurations when approaching 5 hours. This is reasonable because given enough time, a random selection will eventually hit near-optimal points.

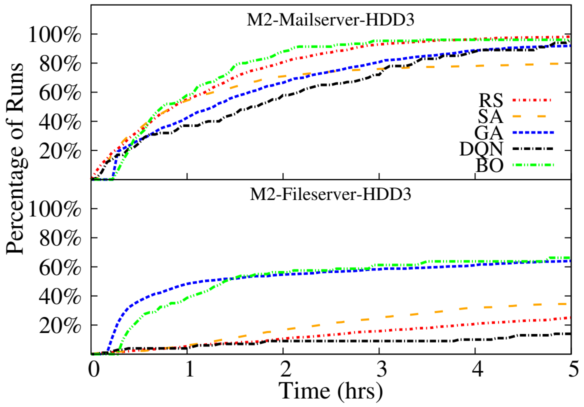


Figure 5: Comparing optimization methods' efficacy in finding near-optimal configurations. The Y-axis shows the percentage of total runs (1,000) that found near-optimal configurations within certain time (X-axis).

However, when conducting the same experiments under *M3-Fileserver-HDD3*, it becomes more difficult to find near-optimal configurations. GA and BO are still the best, though only 65% of their runs were able to find near-optimal configurations within 5 hours. SA, RS, and DQN have a probability of lower than 40% to do so, with DQN perform the worst. This is because the global optimum under *M2-Fileserver-HDD3* is a Btrfs configuration (see Table 4). It is more difficult for optimization algorithms to pick such configurations for the following reasons: 1) Few Btrfs configurations reside in the neighborhood of the default Ext4 configurations; 2) Fewer than 2% of all valid configurations are Btrfs ones, which make them less likely to be selected through mutation; 3) Only 0.2% of all configurations are considered near-optimal, compared with 4.5% in **Mailserver-HDD3**.

The above results focused on finding near-optimal configurations. However, another important aspect to compare is the system's performance *during* the auto-tuning process. This is especially important if the targeted system is deployed and online. Some randomness (exploration) is necessary when searching a complex parameter space, but ideally optimization algorithms should spend less time on bad configurations. To compare this, in Figure 6 we plotted the instantaneous throughput (Y-axis) over time (X-axis) for one run with each method under *M2-Mailserver-HDD3*.

BO and GA are still the best two methods in terms of instantaneous throughput. During the tuning process, occasionally they pick a worse configuration than the current one. However, they both possess the ability to quickly discard these unpromising configurations. GA achieves this by assigning the probability of surviving to next generation based on the fitness values (i.e., throughput). Configurations with low throughput values have a lower chance to be picked as parents, and thus their genes (parameter values) have a lower chance of appearing in configurations of the next generation (i.e., "survival of the fittest"). The reason for stable instantaneous throughputs with BO is that it uses an intelligent acquisition function to guide the selection of the next generation, with the goal of maximizing the potential gain; this makes BO less likely to choose a bad configuration. In contrast, SA performs poorly possibly because it lacks a history to guide the exploitation and exploration phases, and only uses its neighborhood information (and current temperature) to pick the next configuration. DQN shows similar results with RS, which is likely caused by the fact that DQN was originally designed as an agent interacting with an unknown environment, and thus a lot of exploration (randomness) occurs in the training phase [55,85].

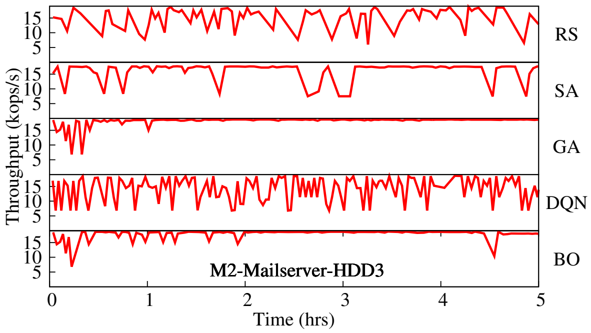


Figure 6: Comparing optimization methods' instantaneous performance (Y-axis) over time (X-axis).

In conclusion, our results demonstrated that the efficacy of different optimization algorithms vary a lot while applied in auto-tuning storage systems. The trade-off among *exploitation*, *exploration*, and *history* plays an important role in find near-optimal configurations efficiently. However, a well-known problem for many optimization techniques is that their performance depend heavily on hyper-parameter settings. Some of our observations may only apply to our specific settings and search spaces. This paper's main goal is not to provide guidelines on which methods are more suitable for auto-tuning storage systems; rather, we focused on comparing multiple methods and understanding their efficacy under different conditions.

4.3 Impact of Hyper-Parameters

Many optimization methods' efficacy depend on the specific hyper-parameter settings, and choosing the right hyper-parameters has caused headache to researchers for a long time [7, 8]. In this section we use GA as a case study, and show the impact of one hyper-parameter, the *mutation rate*, on auto-tuning results. The mutation rate controls the probability of randomly mutating one parameter to a different value, and aligns with the idea of *exploration*, as per §2.4.

Figure 7 shows the results from 7 sets of GA exper-

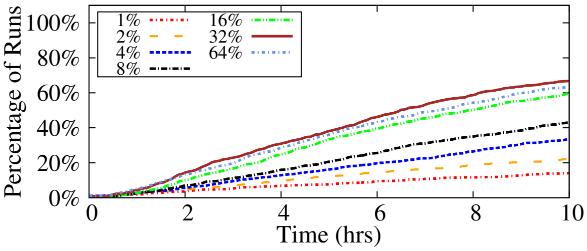


Figure 7: Impact of mutation rates on GA.

iments with different mutation rates (from 1% to 64%) under *M2-Mailserver-HDD3*. Each experiment was repeated for 1,000 runs. It is similar to Figure 5, but with the goal of finding near-optimal configurations whose throughput values are higher than **99.5%** of the global optimal. This makes the optimization more challenging, as GA already performs quite well on easier tasks (§4.2). As shown in the figure, when increasing the mutation rate, GA has a higher probability to converge to near-optimal configurations within a shorter time period. This is because GA works by identifying promising combination of alleles (parameter values) for the subset of dominant genes (parameters). We define dominant parameters as those having a higher impact on performance than all others. A higher mutation rate means a higher chance of exploration, and thus finding combinations of well-performing alleles for the dominant genes within a shorter time. We explain this effect more in §4.4. However, a mutation rate of 64% actually performs worse than 32%. This is because in order to reach near-optimal configurations, GA needs both exploration and exploitation. Exploration lets GA identify processing subspaces (i.e., combinations of certain parameter values) while exploitation helps GA search within promising subspaces. In this case, with a mutation rate of 64%, GA spends too much time on exploration (too much randomness), resulting in fewer chances for exploitation.

4.4 Peering into the Black Box

Despite some successful applications of black-box optimization on auto-tuning system parameters, few have explained how and why some techniques work better than others for certain problems. Here we take the first step towards unpacking the "black box" and provide some insights into their internals based on our evaluation results and storage domain knowledge.

Our attempts for explanations stem from a somewhat unexpected but beneficial behavior of GA in the experiments. We found that as GA runs, there is often a small set of alleles (parameter values) that dominate the current population and are unlikely to change. We present and explain this observation in Figure 8. The experiment was conducted on a parameter space consisting of 2,208 Ext3 configurations under *M2-Fileserver-SSD*. The X-axis shows 5 genes (parameters) separated by red gridlines, while one column represents one allele (parameter value). The parameters are denoted with their abbreviations from Table 2. The Y-axis shows the generation number, and we plotted only the first 8 generations. Cells were colored based on the number of alleles in each generation. More frequent alleles are colored with darker colors. In the first generation, the gene's alleles (parameter values) were quite diverse. For example, there were 3 alleles (1K, 2K, 4K) for the *Block Size* gene, and 3 alleles (journal, ordered, writeback) for the *Journal Option* gene. However, the diversity of alleles decreased in later generations, and several genes began to dominate and even converged to a single allele. For the *Block Size* gene, only the 4K allele survived and other two became extinct. Since GA was proposed by simulating the process of natural selection, where alleles with better fitness are more likely to survive, this suggests that GA works by identifying the combination of good alleles (storage parameter values), and producing offspring with these alleles. As shown in Figure 8, in the 8^{th} generation, all configurations have a *Block Size* of 4K and *Journal Option* of writeback.

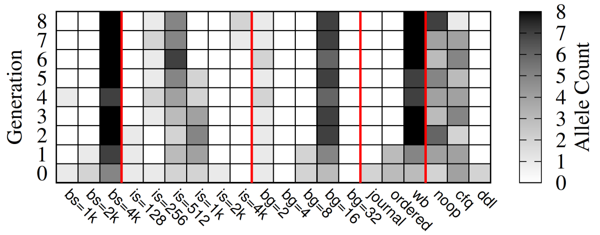


Figure 8: Number of alleles in the first 8 GA generations, with more frequent ones colored with darker colors.

To confirm the above observations, in Figure 9 we plotted all Ext3-SSD configurations, with one dot corresponding to one configuration. Configurations are separated based on the *Journal Option*, shown as the X-axis, and colored based on their *Block Size*. To clearly see all points within each X-axis section, we ordered configurations by their unique identification number in our database. The Y-axis represents throughput values. This resulted in the formation of nine "clusters" on the graph, each corresponding to a fixed ⟨*Journal Option*, *Block Size*⟩ pair. We can see that configurations with *data=ordered* tend to produce higher throughput than those with *data=journal*, and *data=writeback* produces the best throughput. This is somewhat expected from a storage point of view, as Ext3's more fault tolerant journal option (*data=journal*) may hurt throughput by writing data as well as meta-data to the journal first. Moreover, among journal configurations with *data=writeback*, those with a 4K *Block Size* turn out to produce the highest throughput. This aligns with our observation from Figure 8 that GA works by identifying a subset of genes that have a greater impact

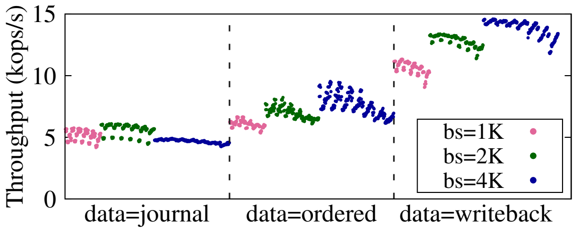


Figure 9: Scatter plot for all Ext3-SSD configurations, with one dot corresponding to one configuration.

on performance—*Block Size* and *Journal Option*—and finding the best alleles for them ([*4K, data=writeback*]).

Based on these observations, one interesting question to ask is whether the conclusion that a subset of parameters have greater impact on performance than other parameters, also holds for other file systems and workloads. To answer this question, we quantified the correlation between each parameter and throughput values. As most of our parameters are categorical or discrete numeric, whereas the throughput is continuous, we took a common approach to quantify the correlation between categorical and continuous variables [14]. We illustrate with the *Block Size* parameter as an example. Since it can take 3 values, we convert this parameter to three binary variables x_1, x_2, and x_3. If the *Block Size* is 1K, we assign $x_1 = 1$ and x_2 and x_3 are set to 0. Let Y represent the throughput values. We then do a linear regression with ordinary least squares (OLS) on Y and x_1, x_2, x_3. R^2 is a common metric in statistics to measure how the data fits a regression line. In our approach, R^2 actually quantifies the correlation between the selected parameter and throughput. We consider $R^2 > 0.6$ as an indication that the parameter has significant impact on performance, as is common in statistics [14]. The same calculation is applied to all parameters for each file system under *M2-Fileserver-SSD* and *M2-Dbserver-SSD*. Parameters with the highest R^2 values are colored in yellow background in Table 5. If all R^2 values are below 0.6, we simply leave the entries blank, meaning no highly correlated parameters were found. To find the second important parameter, the same process is applied to the remaining parameters, but with the value of the most important one fixed (to isolate its effect on the remaining parameters' importance). Taking Ext4 under *M2-Fileserver-SSD* as an example, we calculate R^2 values for all other parameters among configurations with the same *Journal Option*. For one parameter, 3 *Journal Options* lead to three R^2 values; we then take the maximum one as the R^2 value for this parameter. We color the parameter with the highest R^2 in Table 5 with a green background.

We can see that the correlated parameters are quite diverse, and depend a lot on file systems. For example, under *M2-Fileserver-SSD*, the two most important parameters for Ext3 (in descending order) are *Journal Option* and *Block Size*; this aligns with our observation in Figures 8 and 9. However, for Reiserfs, the top 2 changes to *I/O Scheduler* and *Journal Option*. Interestingly, all parameters for Btrfs come with low R^2 values, which indicates that no parameter has significant impact on system performance under *M2-Fileserver-SSD* with Btrfs. Correlation of parameters can also depend on the workloads. For instance, the two dominant parameters for XFS under *M2-Fileserver-SSD* are *Block Size* and *Allocation Group*. When the workload changes to *M2-Dbserver-SSD*, all parameters for XFS seem to have minor impact on performance. In this paper we are isolating the impact of each parameter, thus assuming that their effect on throughput is independent. Note that the above observations are made based on our collected data sets, and might change on different workloads and hardware. However, our methodology is generally applicable. Moreover, this paper's main goal is *not* to suggest guidelines on what specific storage configurations to deploy under certain workloads; rather, we focus on comparing multiple optimization methods and providing insights into their operation.

WL-Dev	FS	BS	IS	BG	JO	AO	SO	I/O
File-SSD	**Ext2**	-	-	-	-	-	-	0.68
	Ext3	0.84	-	-	0.90	-	-	-
	Ext4	0.92	-	-	0.99	-	-	-
	XFS	0.94	-	0.82	-	-	-	-
	Btrfs	-	-	-	-	-	-	-
	Nilfs2	0.99	-	-	-	-	-	0.94
	Reiserfs	-	-	-	0.74	-	-	0.99
Db-SSD	**Ext2**	-	-	-	-	-	-	-
	Ext3	0.72	-	-	0.96	-	-	-
	Ext4	-	-	-	0.96	0.68	-	-
	XFS	-	-	-	-	-	-	-
	Btrfs	-	-	-	-	-	-	-
	Nilfs2	0.62	-	-	-	-	-	0.80
	Reiserfs	-	-	-	0.99	-	-	-

Table 5: Importance of parameters (measured by R^2), with the most important one colored in yellow and second in green.

The fact that parameters have varied impact on performance can also help explain the auto-tuning results in §4.2. Although our parameter space comes with 8 parameters, only a subset of them are highly correlated with performance. As long as the optimization algorithm identifies the "correct" combination of values for these dominant parameters, it will be able to find a near-optimal configuration. Similar behavior has been reported in hyper-parameter optimization problems [7]. For the experiments shown in Figure 4, near-optimal configurations take up 4.5% of the whole search space. Random Search (RS) needs to hit only one of them to achieve good auto-tuning results. GA's efficacy comes from assigning a higher chance of survival to configurations with a certain combination of values for the dominant parameters. BO stores its previous search experience (history) in a probabilistic surrogate model that it is building, which eventually encodes the combination of dominant parameter values that can result in good throughput values. SA does not work as well because it lacks history information to identify the dominant parameters: it wastes time changing less useful parameters and converges slowly. Similarly, DQN also spends lots of its effort on exploring unpromising spaces, which slows its ability to find near-optimal configurations.

5 Limitations and Future Work

In this paper we provided the first comparative analysis of applying multiple optimization methods on auto-tuning storage systems. However, auto-tuning is a complex topic and more effort is required. We list some limitations of this work and our future research directions below. ■ **(1)** We plan to extend the scope of evaluation with more complex workloads and search spaces. We will investigate more techniques, such as experiment design [80], as will as the impact of algorithm hyper-parameter settings [8]. ■ **(2)** We plan to improve traditional optimization techniques with new features, such as penalty functions to cope with costly parameter changes, stopping/restarting criteria, workload identification, handling noisy and unstable results [12], etc., which makes auto-tuning algorithms more robust to environment changes and more generally applicable in production systems.

6 Related Work

Auto-tuning computer systems. In recent years, several attempts were made to automate the tuning of storage systems. Strunk et al. [71] proposed to use utility functions combining different system metrics and applied GA to automate storage system provisioning. Babak et al. [5] utilized GA to optimize I/O performance of HDF5 applications. GA has also been applied for storage recovery problems [38]. More recently, Deep Q-Networks has been successfully applied in optimizing performance for Lustre [85]. Auto-tuning is also a hot topic in other computer systems: Bayesian Optimization was applied to find near-optimal configurations for databases [78] and Cloud VMs [3]. Other applied techniques include Evolutionary Strategies [62], Simulated Annealing [26, 35], Tabu Search [63], and more. However, previous work all focused on one or a few techniques. One contribution of our work is to provide the first comparative study of multiple, applicable optimization methods on their efficacy in auto-tuning storage systems from various aspects. We also provide some insights into the working mechanism of auto-tuning.

Hyper-parameter tuning. Evolutionary Algorithms [59], Reinforcement Learning [6], and Bayesian Optimization [22] have been applied to hyper-parameter optimization for ML algorithms. Bergstra and Bengio [8] found that randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid, and explained the cause as the objective function having a low effective dimensionality. Another direction of research focuses on eliminating all hyper-parameters and tries to propose non-parametric versions of optimization methods. Examples of this include GA [31,51] and BO [68].

7 Conclusions

Optimizing storage systems can provide significant benefits especially in improving I/O performance. Alas, storage systems are getting more complex, contain many parameters and an immense number of possible configurations; manual tuning is therefore impractical. Worse, many of those parameters are non-linear or non-numeric; traditional linear-regression-based optimization techniques do not work well for such problems. Several efforts were made to apply black-box optimization techniques to auto-tune storage systems, but they all used only one or few techniques. In this work, we performed the first comparative study, and offered the following four contributions. **(1)** We evaluated *five* popular but different auto-tuning techniques, varied some of their hyper-parameters, and applied them to storage and file systems. **(2)** We show that the speed at which the techniques can find optimal or near-optimal configurations (in terms of throughput) depends on the hardware, software, and workload; this means that no single technique can "rule them all." **(3)** We explain why some techniques appear to work better than others. **(4)** For more than two years, we have collected a large data set of over 450,000 data points; this data set was used in this study and we plan to release it.

Acknowledgments

We thank the anonymous ATC reviewers and our shepherds Keith Smith and Siddhartha Sen for their valuable comments. This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; NSF awards CNS-1251137, CNS-1302246, CNS-1305360, CNS-1622832, CNS-1650499, and CNS-1730726; and ONR award N00014-16-1-2264.

References

[1] Abutalib Aghayev, Mansour Shafaei, and Peter Desnoyers. Skylight—a window on shingled disk operation. *Trans. Storage*, 11(4):16:1–16:28, October 2015.

[2] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. Evolving ext4 for shingled

disks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 105–120, Santa Clara, CA, February/March 2017. USENIX Association.

[3] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482. USENIX Association, 2017.

[4] Terry Anderson. *The theory and practice of online learning*. Athabasca University Press, 2008.

[5] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. Taming parallel i/o complexity with auto-tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 68:1–68:12, New York, NY, USA, 2013. ACM.

[6] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[7] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[8] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems 24*, pages 2546–2554, 2011.

[9] Christopher M Bishop. *Pattern Recognition and Machine Learning*, volume 1. Springer New York, 2006.

[10] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[11] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[12] Zhen Cao, Vasily Tarasov, Hari Raman, Dean Hildebrand, and Erez Zadok. On the performance variation in modern storage stacks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 329–343, Santa Clara, CA, February/March 2017. USENIX Association.

[13] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings to the First Dutch International Symposium on Linux*, Amsterdam, Netherlands, December 1994.

[14] George Casella and Roger L Berger. *Statistical Inference*, volume 2. Duxbury Pacific Grove, CA, 2002.

[15] Vladimír Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[16] Yvonne Coady, Russ Cox, John DeTreville, Peter Druschel, Joseph Hellerstein, Andrew Hume, Kimberly Keeton, Thu Nguyen, Christopher Small, Lex Stein, and Andrew Warfield. Falling off the cliff: When systems go nonlinear. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10*, HOTOS'05, 2005.

[17] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. Boat: Building auto-tuners with structured bayesian optimization. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 479–488. International World Wide Web Conferences Steering Committee, 2017.

[18] Kenneth Alan De Jong. *Analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan, Ann Arbor, Ann Arbor, MI, USA, 1975.

[19] Y. Diao, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano. Using MIMO linear control for load balancing in computing systems. In *2004 American Control Conferences*, 2004.

[20] Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1(4):28–39, 2006.

[21] Fred Douglis, Deepti Bhardwaj, Hangwei Qian, and Philip Shilane. Content-aware load balancing for distributed backup. In *Large Installation System Administration Conference (LISA)*, 2011.

[22] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, volume 10, 2013.

[23] A.E. Eiben and C.A. Schippers. On evolutionary exploration and exploitation. *Fundam. Inf.*, 35(1-4):35–50, January 1998.

[24] Ext4. *http://ext4.wiki.kernel.org/*.

[25] Filebench, 2016. *https://github.com/filebench/filebench/wiki*.

[26] Terry L Friesz, Hsun-Jung Cho, Nihal J Mehta, Roger L Tobin, and G Anandalingam. A simulated annealing approach to the network design problem with variational inequality constraints. *Transportation Science*, 26(1):18–26, 1992.

[27] Fred Glover and Manuel Laguna. *Tabu Search*. Springer, 2013.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. *http://www.deeplearningbook.org*.

[29] Gradient descent. *https://en.wikipedia.org/wiki/Gradient_descent*.

[30] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The tail at store: a revelation from millions of hours of disk and ssd deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 263–276, 2016.

[31] Georges R Harik and Fernando G Lobo. A parameter-less genetic algorithm. In *GECCO*, volume 99, pages 258–267, 1999.

[32] Weiping He and David H.C. Du. Smart: An approach to shingled magnetic recording translation. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 121–134, Santa Clara, CA, February/March 2017. USENIX Association.

[33] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tibury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[34] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U. Michigan Press, 1975.

[35] Young-Jae Jeon, Jae-Chul Kim, Jin-O Kim, Joong-Rin Shin, and Kwang Y Lee. An efficient simulated annealing algorithm for network reconfiguration in large-scale distribution systems. *Power Delivery, IEEE Transactions on*, 17(4):1070–1078, 2002.

[36] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating system profiling via latency analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 89–102, Seattle, WA, November 2006. ACM SIGOPS.

[37] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Trans. Storage*, 1(4), 2005.

[38] Kimberly Keeton, Dirk Beyer, Ernesto Brau, Arif Merchant, Cipriano Santos, and Alex Zhang. On the road to recovery: Restoring data after disasters. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys'06, pages 235–248, New York, NY, USA, 2006. ACM.

[39] James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[40] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 33–45, Berkeley, CA, 2014. USENIX.

[41] S. Kirkpatrick, C D. Gelatt, M. P Vecchi, et al. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[42] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[43] Ernest Bruce Lee and Lawrence Markus. Foundations of optimal control theory. Technical report, DTIC Document, 1967.

[44] H. D. Lee, Y. J. Nam, K. J. Jung, S. G. Jung, and C. Park. Regulating I/O performance of shared storage with a control theoretical approach. In *NASA/IEEE Conference on Mass Storage Systems and Technologies (MSST)*. IEEE Society Press, 2004.

[45] Yin Li, Hao Wang, Xuebin Zhang, Ning Zheng, Shafa Dahandeh, and Tong Zhang. Facilitating magnetic recording technology scaling for data center hard disk drives through filesystem-level transparent local erasure coding. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 135–148, Santa Clara, CA, February/March 2017. USENIX Association.

[46] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[47] Z. Li, M. Chen, A. Mukker, and E. Zadok. On the trade-offs among performance, energy, and endurance in a versatile hybrid drive. *ACM Transactions on Storage (TOS)*, 11(3), July 2015.

[48] Z. Li, K. M. Greenan, A. W. Leung, and E. Zadok. Power consumption in enterprise-scale backup storage systems. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

[49] Z. Li, R. Grosu, K. Muppalla, S. A. Smolka, S. D. Stoller, and E. Zadok. Model discovery for energy-aware computing systems: An experimental evaluation. In *Proceedings of the 1st Workshop on Energy Consumption and Reliability of Storage Systems (ERSS'11)*, Orlando, FL, July 2011.

[50] Z. Li, A. Mukker, and E. Zadok. On the importance of evaluating storage systems' $costs. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, 2014.

[51] Fernando G Lobo and David E Goldberg. The parameter-less genetic algorithm in practice. *Information Sciences*, 167(1):217–232, 2004.

[52] Peter Merz. Memetic algorithms for combinatorial optimization problems: Fitness landscapes and effective search strategies, 2001.

[53] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2015)*, pages 177–190, Portland, OR, June 2015. ACM.

[54] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[55] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[56] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[57] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD failures in datacenters: What? when? and why? In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '16)*, pages 7:1–7:11, Haifa, Israel, May 2016. ACM.

[58] Christian S. Perone. Pyevolve: A python open-source framework for genetic algorithms. *SIGEVOlution*, 4(1):12–20, November 2009.

[59] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[60] H. Reiser. ReiserFS v.3 whitepaper. *http://web.archive.org/web/20031015041320/http://namesys.com/*.

[61] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[62] Anooshiravan Saboori, Guofei Jiang, and Haifeng Chen. Autotuning configurations in distributed systems for performance improvements using evolutionary strategies. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, ICDCS '08, pages 769–776, Washington, DC, USA, 2008. IEEE Computer Society.

[63] Sadiq M Sait, Mahmood R Minhas, Junhaid Khan, et al. Performance and low power driven vlsi standard cell placement using tabu search. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 1, pages 372–377. IEEE, 2002.

[64] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, pages 231–244, Monterey, CA, January 2002. USENIX Association.

[65] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 67–80, Santa Clara, CA, February 2016. USENIX Association.

[66] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating performance and energy in file system server workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 253–266, San Jose, CA, February 2010. USENIX Association.

[67] Burr Settles. *Active Learning*. Morgan & Claypool Publishers, 2012.

[68] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[69] Shai Shalev-Shwartz. Online learning and online convex optimization. *Foundations and Trends in Machine Learning*, 4(2):107–194, 2011.

[70] Scikit-Optimize. *https://scikit-optimize.github.io/*.

[71] John D. Strunk, Eno Thereska, Christos Faloutsos, and Gregory R. Ganger. Using utility to provision storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 313–328, Berkeley, CA, USA, 2008. USENIX Association.

[72] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[73] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 1–14, San Diego, CA, January 1996.

[74] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking file system benchmarking: It *is* rocket science. In *Proceedings of HotOS XIII:The 13th USENIX Workshop on Hot Topics in Operating Systems*, Napa, CA, May 2011.

[75] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[76] TensorFlow. *https://www.tensorflow.org/*.

[77] Stephen Tweedie. Ext3, journaling filesystem. In *Ottawa Linux Symposium*, July 2000. *http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html*.

[78] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, 2017.

[79] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Comput. Surv.*, 45(3):35:1–35:33, July 2013.

[80] Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation*, pages 363–378, 2016.

[81] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with cart models. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems. (MASCOTS)*, pages 588–595, 2004.

[82] Watts up? PRO ES power meter. *www.wattsupmeters.com/secure/products.php*.

[83] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, Dec 2010.

[84] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 287–296, New York, NY, USA, 2004. ACM.

[85] Oceane Bel Ethan L. Miller Darrell D. E. Long Yan Li, Kenneth Chang. Capes: Unsupervised system performance tuning using neural network-based deep reinforcement learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, 2017.

[86] Yang Yu, Hong Qian, and Yi-Qi Hu. Derivative-free optimization via classification. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2286–2292. AAAI Press, 2016.

[87] Erez Zadok, Aashray Arora, Zhen Cao, Akhilesh Chaganti, Arvind Chaudhary, and Sonam Mandal. Parametric optimization of storage systems. In *HotStorage '15: Proceedings of the 7th USENIX Workshop on Hot Topics in Storage*, Santa Clara, CA, July 2015. USENIX, USENIX.

HeavyKeeper: An Accurate Algorithm for Finding Top-*k* Elephant Flows

Junzhi Gong
Peking University

Tong Yang*
Peking University

Haowei Zhang
Peking University

Hao Li
Peking University

Steve Uhlig
UK & Queen Mary, University of London

Shigang Chen
University of Florida

Lorna Uden
Staffordshire University

Xiaoming Li
Peking University

Abstract

[1] Finding top-*k* elephant flows is a critical task in network traffic measurement, with many applications in congestion control, anomaly detection and traffic engineering. As the line rates keep increasing in today's networks, designing accurate and fast algorithms for online identification of elephant flows becomes more and more challenging. The prior algorithms are seriously limited in achieving accuracy under the constraints of heavy traffic and small on-chip memory in use. We observe that the basic strategies adopted by these algorithms either require significant space overhead to measure the sizes of all flows or incur significant inaccuracy when deciding which flows to keep track of. In this paper, we adopt a new strategy, called *count-with-exponential-decay*, to achieve space-accuracy balance by actively removing small flows through decaying, while minimizing the impact on large flows, so as to achieve high precision in finding top-*k* elephant flows. Moreover, the proposed algorithm called HeavyKeeper incurs small, constant processing overhead per packet and thus supports high line rates. Experimental results show that HeavyKeeper algorithm achieves 99.99% precision with a small memory size, and reduces the error by around 3 orders of magnitude on average compared to the state-of-the-art.

1 Introduction

1.1 Background and Motivation

Finding the largest *k* flows, also referred to as the top-*k* elephant flows, is a fundamental network management function, where a flow's ID is usually defined as a combination of certain packet header fields, such as source IP address, destination IP address, source port, destination port, and protocol type, and the size of a flow is defined as the number of packets of the flow. Elephant flows contribute a large portion of network traffic. Many management applications can benefit from a function that can find them efficiently, such as congestion control by dynamically scheduling elephant flows [1], network capacity planning [2], anomaly detection [3], and caching of forwarding table entries [4]. Such a function also has applications beyond networking in areas such as data mining [5–7], information retrieval [8], databases [9], and security [10, 11].

In real network traffic, it is well known that the distribution of flow sizes (the number of packets in a flow), is highly skewed [12–21], *i.e.*, the majority are mouse flows, while the minority are elephant flows. most flows are small while a few flows are very large. The small flows are usually called *mouse flows*, while the large ones are called *elephant flows*.

Finding the top-*k* elephant flows (or top-*k* flows for short) in high-speed networks is a challenging task. [22] Extremely high line rates of modern networks make it practically impossible to accurately track the information of all flows. Consequently, approximate methods have been proposed in the literature and gained wide acceptance [14, 23–27]. In order to keep up with the line rates, these algorithms are expected to use on-chip memory such as SRAM whose latency is around 1ns [28, 29], in contrast to a latency of around 50ns when off-chip DRAM is used [29]. However, on-chip memory is small. Adding to the challenge, it is highly desirable to keep per-packet processing overhead small and constant, which helps pipelining.

Traditional solutions to finding the top-*k* flows follow two basic strategies: *count-all* and *admit-all-count-some*. The count-all strategy relies on a sketch (*e.g.*, CM sketch [14]) to measure the sizes of all flows, while us-

[1] Co-primary authors: Junzhi Gong and Tong Yang. Corresponding author: Tong Yang (yangtongemail@gmail.com). Junzhi Gong, Haowei Zhang, Hao Li finished this work under the guidance of their supervisor: Tong Yang. This work is supported by Primary Research & Development Plan of China (2016YFB1000304), National Basic Research Program of China (973 Program, 2014CB340405), NSFC (61672061), the OpenProject Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences.

ing a min-heap to keep track of the top-k flows. For each incoming packet, it records the packet in the sketch and retrieves from the sketch an estimate $\hat{n}_i$ for the size of the flow f_i that the packet belongs to. If $\hat{n}_i$ is larger than the smallest flow size in the min-heap, it replaces the smallest flow in the heap by flow f_i. As a large sketch is needed to count all flows, these solutions are not memory efficient.

The *admit-all-count-some* strategy is adopted by Frequent [30], Lossy Counting [26], Space-Saving [24] and CSS [23]. These algorithms are similar to each other. To save memory, Space-Saving only maintains a data structure called Stream-Summary to counts only some flows (m flows). Each new flow will be inserted into the summary, replacing the smallest existing flow. The initial size of the new flow is set as $\hat{n}_{min}+1$, where $\hat{n}_{min}$ is the size of the smallest flow in the summary. By keeping m flows in the summary, the algorithm will report the largest k flows among them, where $m > k$. It assumes every new incoming flow is an elephant, and expels the smallest one in the summary to make room for the new one. But most flows are mouse flows. Such an assumption causes significant error, especially under tight memory (for a limited value of m).

1.2 Our Proposed Solution

In this paper, we propose a new algorithm, HeavyKeeper, based on a different strategy, called *count-with-exponential-decay*, which keeps all elephant flows while drastically reducing space wasted on mouse flows. Unlike *count-all*, our strategy only keeps track of a small number of flows. Unlike *admit-all-count-some*, we do not automatically admit new flows into our data structure and the vast majority of mouse flows will be by-passed. For a small number of mouse flows that do enter our data structure, they will decay away to make room for true elephants. The decay is not uniform for the flows in our data structure. The design of exponential decay is biased against small flows, and it has a smaller impact on larger flows. This design works extremely well with real traffic traces under small memory where the previous strategies will fail.

Main experimental results: As shown in Table 1, when compared with *Space-Saving*, *Lossy counting*, *CSS*, and *CM sketch*, HeavyKeeper achieves 99.99% percent precision, and much smaller error than all of them.

Table 1: Main experimental results.
Precision is defined as the ratio between the number of correctly reported elephant flows and the total number of reported flows.

Algorithm	**Top-k precision**	**Avg. relative error of flow sizes**
Space-Saving [24]	0.27	172.7222
Lossy counting [26]	0.39	54.8440
CSS [23]	0.49	18.9356
CM sketch [14]	0.93	0.2951
HeavyKeeper	0.9999	0.0011

Contributions: This paper makes the following contributions.

1. We propose a new data structure, named HeavyKeeper, which achieves high precision for finding top-k flows, and achieves constant and fast speed as well as high memory efficiency.

2. We develop a mathematical analysis for HeavyKeeper, to theoretically prove its high precision.

3. We conduct extensive experiments on real network streams and synthetic datasets, and results show that HeavyKeeper reduces the error by around 3 orders of magnitude on average compared to the state-of-the-art.

4. We integrate HeavyKeeper and other related algorithms with Open vSwitch (OVS) platform. We also conduct experiments on throughput on OVS platform to show the impact of the algorithms. The results show that HeavyKeeper has little impact on the throughput, while other algorithms decrease the throughput significantly. We release the source code of HeavyKeeper and related algorithms at GitHub [31].

2 Preliminaries

2.1 Problem Statement

Simply speaking, finding top-k flows refers to finding the largest k flows. Let $\mathscr{P} = \mathbb{P}_1, \mathbb{P}_2, \cdots, \mathbb{P}_N$ be a network stream with N packets. Each packet $\mathbb{P}_l$ ($1 \leqslant l \leqslant N$) belongs to a flow f_i, where $f_i \in \mathscr{F} = \{f_1, f_2, \cdots, f_M\}$ and $\mathscr{F}$ is the set of flows. Let n_i be the real flow size of flow f_i in $\mathscr{P}$. We order all flows ($f_1, f_2, \cdots, f_M$) so that $n_1 \geqslant n_2 \geqslant \cdots \geqslant n_M$.

Given an integer k and a network stream $\mathscr{P}$, the output of top-k is a list of k flows from $\mathscr{F}$ with the largest flow sizes, *i.e.*, $f_1, f_2, \cdots, f_k$.

2.2 Prior Art and Limitations

The count-all strategy: As mentioned above, the *count-all* strategy uses sketches (such as the CM sketch [14] or the Count sketch [25]) to record the sizes of all flows, and uses a min-heap to keep track of the top-k flows, including the flow IDs and their flow sizes. Take the CM sketch as an example. It records packets in a CM sketch, consisting of a pool of counters. For each arrival packet, it hashes the packet's flow ID f to d counters and increases these d counters by one. The smallest value of the d counters is used as the estimated size of the flow.

If this estimated flow size is larger than the smallest flow size in the min-heap, we replace the smallest flow in the heap by flow f.

The problem is that all flows are pseudo-randomly mapped to the same pool of counters through hashing. Each counter may be shared by multiple flows, and thus record the sum of sizes of all these flows. Consequently, the CM sketch has an over-estimation problem, which will become severe in a tight memory space where the number of counters is far smaller than the number of flows, resulting in aggressive sharing. In such a case, a small flow may be treated as an elephant flow if all its d counters are shared with real elephant flows.

The admit-all-count-some strategy: As mentioned above, quite a few algorithms use the *admit-all-count-some* strategy, including Frequent [30], Lossy counting [26], and Space-Saving [24], with Space-Saving being the most widely used among them. Take Space-Saving as an example. Recognizing that it is infeasible to count the sizes of all flows, Space-Saving counts only the sizes of some flows in a data structure called Stream-Summary, which incurs $O(1)$ overhead to search or update a flow, or find the smallest flow. The selection of which flows to store in the summary is rather simple: For each arrival packet, if its flow ID is not in the summary, the flow will be admitted into the summary, replacing the smallest existing flow. The new flow's initial size is set to $\hat{n}_{min}+1$, where $\hat{n}_{min}$ is the smallest flow size in the summary before replacement. Therefore, later incoming mouse flows will be largely over-estimated, which is drastically inaccurate. In the end, the largest k flows in the summary will be reported. A recent work CSS [23] is proposed based on Space-Saving. It inherits the above strategy of Space-Saving, and redesigns the data structure of Stream-Summary by using TinyTable [32] to reduce memory usage.

The strategy of *admit-all-count-some* is to admit all new flows while expelling the smallest existing ones from the summary. To give new flows a chance to stay in the summary, their initial flow sizes are set as $\hat{n}_{min}+1$. Such a strategy drastically over-estimates sizes of flows, and we show an example here. Assume $\hat{n}_{min} = 10,000$. Given an new incoming flow, its size will be over-estimated as $10,001$. Early arrived elephant flows with flow sizes less than $10,008$ will be expelled. Therefore, massive mouse flows will cause significant over-estimation errors.

3 The Design of HeavyKeeper

In this section, we present the data structure and algorithm of our HeavyKeeper, and show how to find the top-k flows accurately and efficiently.

3.1 Rationale

We aim to use a small hash table to store all elephant flows. As there are a great number of flows, each bucket of the hash table will be mapped by many flows, and we aim to store only the largest flow with its size, which cannot be achieved with no error when using small memory. To address this problem, we propose a probabilistic method called *exponential-weakening decay*. Specifically, when the incoming flow is different from the flow in the hashed bucket, we decay the flow size with a decay probability, which is exponentially smaller as the flow size grows larger. If the flow size is decayed to 0, it replaces the original flow with the new flow. In this way, mouse flows can easily be decayed to 0, while elephant flows can easily keep stable in the bucket. There are two shortcomings: 1) With a small probability we elect the wrong flow as the largest flow; 2) The stored flow size is a little smaller than the true frequency because of the decay operations. To address these problems, we use multiple hash tables with different hash functions. An elephant flow could be stored in multiple hash tables, we choose the recorded largest size, minimizing the error of flow sizes.

3.2 The HeavyKeeper Structure

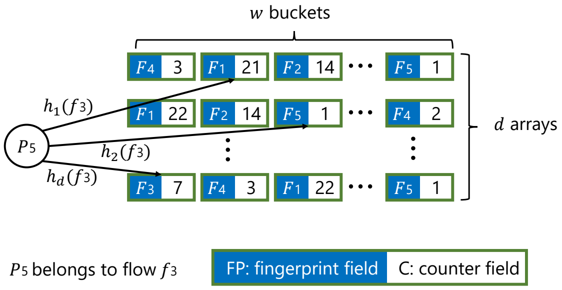


Figure 1: The data structure of HeavyKeeper.

As shown in Figure 1, HeavyKeeper is comprised of d arrays, and each array is comprised of w buckets. Each bucket consists of two fields: a fingerprint field and a counter field.[2] For convenience, we use $A_j[t]$ to represent the t^{th} bucket in the j^{th} array, and use $A_j[t].FP$ and $A_j[t].C$ to represent its fingerprint field and counter field, respectively. Arrays $A_1...A_d$ are associated with hash functions $h_1(.)...h_d(.)$, respectively. These d hash functions $h_1(.)...h_d(.)$ need to be pairwise independent.

Insertion: Initially, all fingerprint fields are *null*, and all counter fields are 0. For each incoming packet $\mathbb{P}_l$ belong-

[2] The fingerprint of a flow is a hash value generated by a certain function (for example, if we use $h_f(.)$ as the fingerprint hash function, the fingerprint of flow f_j is $h_f(f_j)$). Although there can be hash collisions among flows, the probability is quite small. For example, if we set the fingerprint size to 16 bits, and there are 10000 buckets in the array, the probability of fingerprint collisions is $1.52*10^{-3}$.

ing to flow f_i, HeavyKeeper computes the d hash functions, and maps f_i to d buckets $A_j[h_j(f_i)]$ $(1 \leqslant j \leqslant d)$ (one bucket in each array), which we call ***d* mapped buckets** for convenience. As shown in Figure 2, for each mapped bucket, HeavyKeeper applies different strategies for the following three cases:

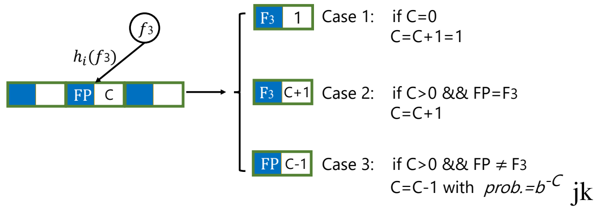


Figure 2: The main insertion cases of HeavyKeeper. Note: 1) $\mathbb{F}_3$ is the fingerprint of flow f_3. 2) $b > 1$ and $b \approx 1$ (*e.g.*, $b = 1.08$). 3) In Case 3, when C is decayed to 0, the fingerprint field will be replaced by $\mathbb{F}_3$, and then counter C is set to 1.

Case 1: When $A_j[h_j(f_i)].C = 0$. It means that no flow has been mapped to this bucket, then HeavyKeeper sets $A_j[h_j(f_i)].FP = \mathbb{F}_i$ and $A_j[h_j(f_i)].C = 1$, where $\mathbb{F}_i$ represents the fingerprint of f_i.

Case 2: When $A_j[h_j(f_i)].C > 0$ and $A_j[h_j(f_i)].FP = \mathbb{F}_i$. It means $A_j[h_j(f_i)].C$ is probably the estimated size of f_i. In this case, HeavyKeeper increments $A_j[h_j(f_i)].C$ by 1.

Case 3: When $A_j[h_j(f_i)].C > 0$ and $A_j[h_j(f_i)].FP \neq \mathbb{F}_i$. It means that $A_j[h_j(f_i)].C$ is not the estimated size of f_i. In here, HeavyKeeper applies the *exponential-weakening decay* strategy to this bucket: it decays $A_j[h_j(f_i)].C$ by 1 with a probability P_{decay}. After decay, if $A_j[h_j(f_i)].C = 0$, HeavyKeeper replaces $A_j[h_j(f_i)].FP$ with $\mathbb{F}_i$, and sets $A_j[h_j(f_i)].C$ to 1. Therefore, as long as flows are mapped to a bucket, its counter field will never be 0.

Query: To query the size of a flow f_i, HeavyKeeper first computes the d hash functions to get d buckets $A_j[h_j(f_i)]$ $(1 \leqslant j \leqslant d)$. Among the d mapped buckets, it chooses those buckets whose fingerprint fields are equal to $\mathbb{F}_i$. It then reports the maximum counter field of those buckets, *i.e.*, $max_{1 \leqslant j \leqslant d}\{A_j[h_j(f_i)].C\}$ where $A_j[h_j(f_i)].FP = \mathbb{F}_i$.

For convenience, for those d mapped buckets of f_i, if $A_j[h_j(f_i)].FP = \mathbb{F}_i$, we say that f_i is ***held*** at bucket $A_j[h_j(f_i)]$. Ignoring the limited impact of fingerprint collisions, we prove that *the reported size for each flow is equal to or smaller than the real flow size* in Section 4.1. If a flow is *held* at no mapped bucket, it reports that it is a mouse flow. If a flow is *held* at multiple buckets, HeavyKeeper reports the maximum counter field.

Decay probability: The decay probability P_{decay} in the exponential-weakening decay strategy is an important parameter. Here, we use the following exponential function to calculate the probability:

$$P_{decay} = b^{-C} \quad (b > 1)$$

where C is the value in the current counter field, and where b ($b > 1$ and $b \approx 1$, *e.g.*, $b = 1.08$) is a pre-defined exponential base number. Therefore, the larger size a flow has, the harder its size is decayed. For elephant flows, it is held at several buckets, and the corresponding counter fields are incremented regularly, while decayed with a very small probability. Therefore, the error rate for estimated sizes of elephant flows is very small.

Note: Our data structure of d arrays may show some similarity with that of CM [14]. But similarity stops there. CM records the sizes of all flows; we record the sizes of a small number of flows. CM does not store flow IDs; we do. CM stores information of each flow in d counters; we keep each flow mostly in one bucket, while d-hashing helps find an empty bucket. CM does not have to worry about the issue of kicking out existing flows to make room for new ones, which is what our exponential delay does.

Example: As shown in Figure 1, given an incoming packet $\mathbb{P}_5$ belonging to flow f_3, we compute the d hash functions to obtain one bucket in each array. In the mapped bucket of the first array, the fingerprint field is not equal to $\mathbb{F}_3$ and the counter field is 21, thus we decay the counter field from 21 to 20 with a probability of 1.08^{-21} (assume $b = 1.08$). In the second mapped bucket, the fingerprint field is not $\mathbb{F}_3$ yet, and with a probability of 1.08^{-1}, we decay the counter field from 1 to 0. If the counter field is decayed to 0, we set the fingerprint field to $\mathbb{F}_3$, and set the counter field to 1. In the last mapped bucket, the fingerprint field is $\mathbb{F}_3$, we increment the counter field from 7 to 8.

Analysis: HeavyKeeper uses fingerprint to identify and keep elephant flows. If a mouse flow with a small flow size is held at a bucket, it will be replaced by other flows mapped to this bucket soon, because each flow mapped to this bucket with a different fingerprint will decay the counter field with a high probability ($b^{-C} \to 1$ when C is small). If an elephant flow is held at a bucket, the corresponding counter field can easily be incremented to a large value since elephant flows have many incoming packets. Moreover, the decay probability becomes very small ($b^{-C} \to 0$ when C is large) as the counter field increases to a large value. Therefore, mouse flows can hardly be held in HeavyKeeper for a long time, and thus have a large probability to be *passers-by* of HeavyKeeper. However, elephant flows can keep stable in HeavyKeeper, and the estimated sizes of elephant flows are accurate.

3.3 Basic Version for Finding the Top-*k* Elephant Flows

To find top-*k* elephant flows, our basic version just uses a HeavyKeeper and a min-heap. The min-heap is used to store the IDs and sizes of top-*k* flows. For each incoming packet $\mathbb{P}_l$ belonging to flow f_i, we first insert it into HeavyKeeper. Suppose that HeavyKeeper reports the size of f_i as $\hat{n}_i$. If f_i is already in the min-heap, we update its estimated flow size with $max(\hat{n}_i, min\_heap[f_i])$, where $min\_heap[f_i]$ is the recorded size of f_i in min-heap. Otherwise, if $\hat{n}_i$ is larger than the smallest flow size which is in the root node of the min-heap, we expel the root node from the min-heap, and insert f_i with $\hat{n}_i$ into the min-heap. To query top-k flows, we simply report the k flows in the min-heap with their estimated flow sizes.

3.4 Optimizations

In this section, we propose further optimization methods to avoid accidental errors and improve speed.

Optimization I: Fingerprint Collisions Detection.
Problems: Assume that there is a bucket in HeavyKeeper where flow f_i is held, and a mouse flow f_j mapped to the same bucket has the same fingerprint as f_i, *i.e.*, $\mathbb{F}_i = \mathbb{F}_j$ due to hash collisions. Then, the mouse flow f_j is also held at this bucket, and its estimated size is drastically over-estimated. In the worst case, if flow f_j has a fingerprint collision in all d arrays, the mouse flow f_j will probably be inserted into the min-heap. It can hardly be expelled due to its drastically over-estimated size. To address this problem, we propose a solution based on the following Theorem.

Theorem 1. *When there is no fingerprint collision, after a flow f_i is inserted into HeavyKeeper, if its estimated size $\hat{n}_i$ is larger than n_{min}, then we must have*

$$\hat{n}_i = n_{min} + 1$$

The proof of this Theorem is not hard to derive and we skip it due to space limitations.

Solution: Based on Theorem 1, if f_i is not in the min-heap but $\hat{n}_i > n_{min} + 1$, then f_i is a mouse flow whose size is drastically over-estimated due to fingerprint collision. Therefore, we should not insert f_i into the min-heap in this case.

Optimization II: Selective Increment.
Problem: If a flow f_i is not in the min-heap, then the estimated flow size should be no larger than n_{min}. However, due to fingerprint collisions, there could be some mapped buckets of flow f_i where the fingerprint field is $\mathbb{F}_i$ and the counter field is larger than n_{min}. In this case, flow f_i is not the flow that is held at this bucket, and thus increasing the corresponding counter field can only incur extra error.

Solution: In this case, instead of incrementing or decaying the corresponding counter field, we make no change.

Optimization III: Speed Acceleration.
Problem: Our basic version of using the min-heap is the most memory efficient solution. However, the processing speed is limited, because the time complexity for updating and searching a flow in the min-heap is $O(log(k))$ and $O(k)$ respectively, which are time-consuming.

Solution: The min-heap is actually used to record the flow IDs of elephant flows and their estimated flow sizes. In this optimization version, instead of using the min-heap, we use a single array to record the flow IDs. Specifically, we define a flow size threshold η (*e.g.*, $\eta = 1000$). For each incoming flow, if its estimated size is equal to η, we record the flow ID in the array. As we record the fingerprints of elephant flows, the flow size will increases at most by 1 for each incoming packet when assuming there is no fingerprint collision. Therefore, any flow whose estimated size is larger than or equal to η is recorded in this array once in most cases. Further, this optimization of using an array is only suitable for sketches that record flow IDs or fingerprints.

Algorithm 1: Insertion process for finding top-k flows.

```
Input: A packet $\mathbb{P}_l$ belonging to flow $f_i$
$flag \leftarrow false$;
if $f_i \in min\_heap$ then
    $flag \leftarrow true$;
$maxv \leftarrow 0$;
for $j \leftarrow 1$ to $d$ do
    $C \leftarrow A_j[h_j(f_i)].C$;
    if $A_j[h_j(f_i)].FP = \mathbb{F}_i$ then
        if $flag = true$ or $C < min\_heap.n_{min}$ then
            $A_j[h_j(f_i)].C++$;
        $maxv \leftarrow max(maxv, A_j[h_j(f_i)].C)$;
    else
        if $rand(1) < b^{-C}$ then
            $A_j[h_j(f_i)].C--$;
            if $A_j[h_j(f_i)].C = 0$ then
                $A_j[h_j(f_i)].FP \leftarrow \mathbb{F}_i$;
                $A_j[h_j(f_i)].C \leftarrow 1$;
                $maxv \leftarrow max(maxv, 1)$;
if $flag = true$ then
    $min\_heap[f_i] \leftarrow max(maxv, min\_heap[f_i])$;
else
    if $min\_heap$ has empty buckets or $maxv - n_{min} = 1$ then
        $min\_heap.insert(f_i)$;
```

3.5 Final Version

Based on the basic version, we propose the common final version using the first two optimization methods, and propose the accelerated final version using the third optimization methods. The insertion and query processes of the common final version of our algorithm are as follows (see pseudo-code in Algorithm 1).

Insertion: All counters and fingerprints in HeavyKeeper and the min-heap are initialized to 0. For each incoming packet $\mathbb{P}_l$ belonging to flow f_i, these are the following three steps for each insertion:

Step 1: Check whether flow f_i is already monitored by the min-heap. For convenience, we use a boolean variable $flag$ to represent the result.

Step 2: Insert f_i into HeavyKeeper. According to Optimization II, for each mapped bucket, if the fingerprint field is equal to $\mathbb{F}_i$, increment the counter field only when $flag = true$ or $C < n_{min}$, where C is the original value in the counter field.

Step 3: Get an estimated size $\hat{n}_i$ of flow f_i from HeavyKeeper. According to Optimization III, if $flag = true$, we update the estimated size of flow f_i in the min-heap with $\hat{n}_i$. If $flag = false$, insert flow f_i into the min-heap with $\hat{n}_i$ in only two cases: 1) the number of flows that are in the min-heap is less than k; 2) $\hat{n}_i = n_{min} + 1$.

Query top-k flows: It reports the k flows recorded in the min-heap and their estimated flow sizes.

Analysis: Since HeavyKeeper achieves very small error rate on the flow size estimation of elephant flows, it can significantly reduce the error in finding top-*k* elephant flows. Furthermore, the first two optimizations reduce the impact of fingerprint collisions, and enhance the precision of finding top-*k* elephant flows and their flow size estimation. The third optimization method has a constant processing time for insertions: 1) For most incoming packets, they are only inserted into HeavyKeeper, which requires d (*e.g.*, $d = 2$) memory accesses. 2) For some packets belonging to elephant flows, they are inserted into both HeavyKeeper and the array. It requires $d+1$ memory accesses in the worst case. Therefore, the time complexity of insertion process is $O(d)$. Therefore, the processing speed of the accelerated final version is fast on average and constant in the worst case.

3.6 Other uses of HeavyKeeper

Besides finding top-*k* flows in a network stream, HeavyKeeper can also perform other tasks in network traffic measurement, such as heavy hitter detection and change detection. Due to space limitations, here we only briefly introduce how to perform these tasks using HeavyKeeper.

Heavy hitter detection: Given a threshold θ, a heavy hitter [13] is a flow whose size $n_i > \theta N$, where N is the number of packets in total. The heavy hitter detection algorithm is very similar to that of finding top-*k* flows. The only difference is that when querying heavy hitters, it reports those flows whose estimated size is larger than θN in min-heap.

Change detection: The network stream is divided into fixed-size time bins. Given a flow, if the difference of its flow sizes in two adjacent time bins is larger than a predefined threshold, then the flow is called a heavy change [13, 33]. We use the very flow ID as the fingerprint of each flow. For two adjacent time bins, we insert their packets into two different HeavyKeepers. By comparing buckets in the corresponding location in the two HeavyKeepers, we obtain the estimated difference of sizes of the flows, and report the heavy changes by checking if the difference is larger than the threshold.

4 Mathematical Analysis

In this section, we first prove that there is no over-estimation in HeavyKeeper, and then derive the formula of its error bounds.

4.1 Proof of No Over-estimation Error of HeavyKeeper

Theorem 2. *Let $n_i(t)$ be the real size of flow f_i at time t, and let $A_j[h_j(f_i)](t).C$ be the counter field of the mapped bucket of flow f_i in the j^{th} array at time t. If there is no fingerprint collision, then*

$$\forall j,t,\ A_j[h_j(f_i)](t).C \leqslant n_i(t) \tag{1}$$

Proof. When $t = 0$, no packet maps into this bucket, so $n_i(0) = 0$ and $A_j[h_j(f_i)](t).C = 0$. Therefore, the theorem holds at time 0. Let's now prove by induction that the theorem holds at any time.

When $t = 0$, the theorem holds.

If the theorem holds when $t = v$, let's prove that the theorem also holds when $t = v+1$. There are three cases when $t = v+1$:

Case 1: The new incoming packet is not mapped to bucket $A_j[h_j(f_i)]$. Then $n_i(v+1) = n_i(v)$ and $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C$. Therefore, $A_j[h_j(f_i)](v+1).C \leqslant n_i(v+1)$.

Case 2: The new incoming packet belongs to flow f_i. Then $n_i(v+1) = n_i(v)+1$ and $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C+1$. Therefore, $A_j[h_j(f_i)](v+1).C \leqslant n_i(v+1)$.

Case 3: The new incoming packet is mapped to bucket $A_j[h_j(f_i)]$ but does not belong to flow f_i. Then $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C$ or $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C-1$, and $n_i(v+1) = n_i(v)$. Therefore, $A_j[h_j(f_i)](v+1).C \leqslant n_i(v+1)$.

Therefore, for any time t,

$$A_j[h_j(f_i)](t).C \leqslant n_i(t)$$

□

4.2 Error Bound of HeavyKeeper

Definition 4.1. *Given a small positive number ε, $Pr\{n_i - \hat{n}_i \geqslant \lceil \varepsilon N \rceil\}$ ($n_i \geqslant \hat{n}_i$) represents the probability that the error of the estimated flow size $n_i - \hat{n}_i$ is larger than εN. If $Pr\{n_i - \hat{n}_i \geqslant \lceil \varepsilon N \rceil\} \leqslant \delta$, the algorithm is said to achieve (ε,δ)-counting.*

(ε,δ)-counting is a metric to evaluate the error rate of the algorithm. Here HeavyKeeper is proved to achieve (ε,δ)-counting, showing that HeavyKeeper achieves a low error rate in estimating the sizes of top-*k* flows.

Theorem 3. *Let's assume that there is no fingerprint collision and the fingerprint of an elephant flow is held at its mapped bucket all the time. Let's focus on one single array of HeavyKeeper. Given a small positive number ε, and an elephant flow f_i whose size is n_i is held at that bucket,*

$$Pr\{n_i - \hat{n}_i \geqslant \lceil \varepsilon N \rceil\} \leqslant \frac{1}{\varepsilon w n_i (b-1)} \tag{2}$$

where w is the width of each array, N the total number of packets, and b the exponential base.

Proof. Let's focus on the j^{th} array. Flow f_i is correctly reported, so at the end, the fingerprint of flow f_i is held in the $h_j(f_i)^{th}$ bucket of the j^{th} array. Let $I_{i,j,i'}$ be a binary random variable, defined as

$$I_{i,j,i'} = \begin{cases} 0 & (f_i = f_{i'}) \vee (h_j(f_i) \neq h_j(f_{i'})) \\ 1 & (f_i \neq f_{i'}) \wedge (h_j(f_i) = h_j(f_{i'})) \end{cases} \tag{3}$$

$I_{i,j,i'} = 1\ iff$ different flows f_i and $f_{i'}$ are held at the same bucket in the j^{th} array. We define random variable $X_{i,j}$ as:

$$X_{i,j} = \sum_{v=1}^{M} I_{i,j,i'} n_{i'} \tag{4}$$

$X_{i,j}$ represents the sum of the sizes of the flows held at the same bucket as flow f_i, except for the size of flow f_i itself. Assume that for each incoming packet, if it belongs to flow f_i, the counter field is incremented by 1; if not, the counter field is decayed with a certain probability. We have

$$n_i - X_{i,j} \leqslant A_j[h_j(f_i)].C \leqslant n_i \tag{5}$$

Specifically, if all packets that do not belong to flow f_i decay the counter field, then $A_j[h_j(f_i)].C = n_i - X_{i,j}$. If those packets do not decay the counter field, then $A_j[h_j(f_i)].C = n_i$. Let's define another random variable $P_{i,j,l}$. Among the $X_{i,j}$ packets defined above, $P_{i,j,l}$ is defined as the probability that the l^{th} packet decays the counter field. Therefore,

$$A_j[h_j(f_i)].C = n_i - \sum_{l=1}^{X_{i,j}} P_{i,j,l} \tag{6}$$

Given a small positive number ε, the following formula based on the Markov inequality holds

$$\begin{aligned} &Pr\{A_j[h_j(f_i)].C \leqslant n_i - \varepsilon N\} \\ &= Pr\{n_i - \sum_{l=1}^{X_{i,j}} P_{i,j,l} \leqslant n_i - \varepsilon N\} \\ &= Pr\{\sum_{l=1}^{X_{i,j}} P_{i,j,l} \geqslant \varepsilon N\} \leqslant \frac{E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})}{\varepsilon N} \end{aligned} \tag{7}$$

Now let's focus on $E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$. Assume that all packets are uniformly distributed, we have the following formula:

$$Pr\{P_{i,j,l} = \frac{1}{b^{\mathrm{C}}}\} = \frac{1}{A_j[h_j(f_i)].C} = \frac{1}{n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})} \tag{8}$$

where $1 \leqslant \mathrm{C} \leqslant n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$. Let β be $n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$ for convenience. As a result,

$$\begin{aligned} E(\sum_{l=1}^{X_{i,j}} P_{i,j,l}) &= \sum_{l=1}^{E(X_{i,j})} E(P_{i,j,l}) \\ &= E(X_{i,j}) \sum_{\mathrm{C}=1}^{\beta} \frac{1}{b^{\mathrm{C}}} \frac{1}{\beta} = \frac{E(X_{i,j})}{\beta} \cdot \sum_{\mathrm{C}=1}^{\beta} \frac{1}{b^{\mathrm{C}}} \\ &= \frac{E(X_{i,j})}{\beta} \cdot \frac{\frac{1}{b}(1-(\frac{1}{b})^{\beta})}{1-\frac{1}{b}} \\ &\leqslant \frac{E(X_{i,j})}{n_i b} \cdot \frac{1-(\frac{1}{b})^{n_i}}{1-\frac{1}{b}} = \frac{E(X_{i,j})(1-(\frac{1}{b})^{n_i})}{n_i(b-1)} \end{aligned} \tag{9}$$

Furthermore, for $E(X_{i,j})$, based on Equation 4,

$$E(X_{i,j}) = E(\sum_{v=1}^{M} I_{i,j,i'} n_{i'}) \quad \leqslant \sum_{i'=1}^{M} n_{i'} E(I_{i,j,v}) = \frac{N}{w} \tag{10}$$

Therefore, based on Equation 9,

$$E(\sum_{l=1}^{X_{i,j}} P_{i,j,l}) \leqslant \frac{N(1-(\frac{1}{b})^{n_i})}{w n_i (b-1)} \leqslant \frac{N}{w n_i (b-1)} \tag{11}$$

then

$$\begin{aligned} Pr\{A_j[h_j(f_i)].C \leqslant n_i - \varepsilon N\} &\leqslant \frac{E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})}{\varepsilon N} \\ &\leqslant \frac{N}{\varepsilon N w n_i (b-1)} = \frac{1}{\varepsilon w n_i (b-1)} \end{aligned}$$

Note that for an elephant flow f_i, n_i is very large, and $(\frac{1}{b})^{n_i} \approx 0$. The estimated size of f_i is the maximum value of $A_j[h_j(f_i)].C$, so we have

$$Pr\{n_i - \hat{n}_i \geqslant \lceil \varepsilon N \rceil\} \leqslant Pr\{\hat{n}_i \leqslant n_i - \varepsilon N\} \leqslant \frac{1}{\varepsilon w n_i (b-1)}$$

□

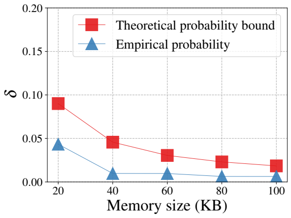


Figure 3: Theoretical bound and empirical probability of HeavyKeeper ($\varepsilon = 2^{-16}$).

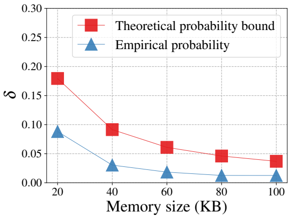


Figure 4: Theoretical bound and empirical probability of HeavyKeeper ($\varepsilon = 2^{-17}$).

To validate the correctness of this error bound, we conduct experiments on the dataset mentioned in Section 5.1. Here, we let $N = 10^7$, $\varepsilon = 2^{-16}$ and 2^{-17}, and vary memory size from 20KB to 100KB. As shown in Figure 3 and Figure 4, the empirical probability of HeavyKeeper is always lower than the theoretical probability bound, confirming the correctness of Theorem 3. Moreover, for the CSS algorithm, achieving such a (ε,δ)-counting requires at least $O(\varepsilon^{-1})$ buckets (*i.e.*, $m = O(\varepsilon^{-1})$), which requires a memory size much larger than 100KB. Therefore, HeavyKeeper is much more memory efficient than CSS.

5 Experimental Results

5.1 Experiment Setup

Platform: Our experiments are run on a server with dual 6-core CPUs (24 threads, Intel Xeon CPU E5-2620 @2 GHz) and 62 GB total system memory. Each core has two L1 caches with 32KB memory (one instruction cache and one data cache) and one 256KB L2 cache. All cores share one 15MB L3 cache.

Dataset:

1) Campus dataset: The first dataset is comprised of IP packets captured from the network of our campus. We rely on the usual definition of a flow, through its 5-tuple, *i.e.*, source IP address, destination IP address, source port, destination port, and protocol type. There are 10M packets in total, belonging to 1M flows. For convenience, we use *campus dataset* to denote this dataset.

2) CAIDA dataset: The second dataset is a CAIDA Anonymized Internet Trace from 2016 [34], made of anonymized IP packets. Each flow in this dataset is identified by the source and destination IP address. We use the first 10M packets, belonging to about 4.2M flows.

3) Synthetic datasets: We generate 10 different synthetic datasets according to a Zipf [35] distribution with different skewness (from 0.3 to 3.0). Each dataset is comprised of 32M packets, belonging to 1 $\sim$ 10M flows depending on the skewness. Each packet is 4 bytes long. The code of the dataset generator is the one from Web Polygraph [36].

Implementation: The implementation of HeavyKeeper is done in C++. We also implemented in C++ the other related algorithms including Space-Saving (SS), Lossy counting (LC), and CM sketch. The source code of CSS was provided by its author [23], and is written in Java. It is much slower than Space-Saving written in C++. Therefore, we do not include CSS in our speed experiments. For Space-Saving, Lossy counting, and CSS, the **number of buckets** m is determined by the memory size, which is usually much larger than k. When querying top-k flows, they report the largest k flows of them. For CM sketch, the size of the heap is k, the number of arrays is 3, and the width of each array is determined by the memory size. In our algorithm, the number of buckets m in Stream-Summary is equal to k, and HeavyKeeper occupies the rest memory size. Here we set $d = 2$, and w depends on the memory size. Both the fingerprint field and the counter field are 16-bit long. For experiments on throughput, we ignore operations on the min-heap for the CM sketch, because we can only record flows whose estimated size is larger than a pre-defined threshold.

5.2 Metrics

Precision: Precision is defined as $\frac{\mathscr{C}}{k}$. Only $\mathscr{C}$ flows belong to the real top-k flows.

Average Relative Error (ARE): ARE is defined as $\frac{1}{|\Psi|}\sum_{f_i\in\Psi}\frac{|\hat{n}_i - n_i|}{n_i}$, where Ψ is estimated set of top-k flows, $\hat{n}_i$ is the estimated size of flow f_i, and n_i is the real size of flow f_i. ARE evaluates the error rate of the estimated flow size reported by the algorithm.

Average Absolute Error (AAE): AAE is defined as $\frac{1}{|\Psi|}\sum_{f_i\in\Psi}|\hat{n}_i - n_i|$, similarly to ARE.

Throughput: We perform insertions of all packets, record the total time used, and calculate the throughput. The throughput is defined as $\frac{N}{T}$, where N is the total number of packets, and T is the total measured time. We use Million of insertions per second (Mps) to measure the throughput.

5.3 Experiments on Precision

To achieve a head-to-head comparison, we use the same memory size for each algorithm. We perform the experiments for varying memory size and k on the campus and CAIDA datasets, and varying skewness on the synthetic datasets. For experiments of varying memory size, we set $k = 100$. For experiments of varying k, we set the memory size to 100KB. For experiments of varying skewness, we set the memory size to 100KB and set $k = 1000$.

Precision vs. memory size: As shown in Figure 5, for the campus dataset, when memory size is 10KB, the precision of Space-Saving, Lossy counting, CSS, and CM

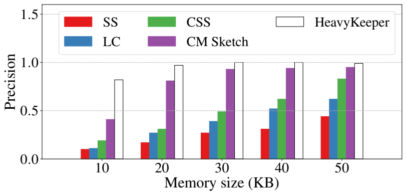


Figure 5: Precision vs. memory size (Campus).

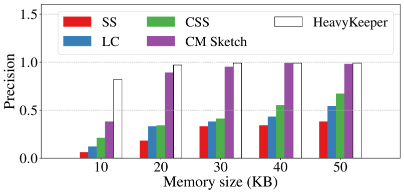


Figure 6: Precision vs. memory size (CAIDA).

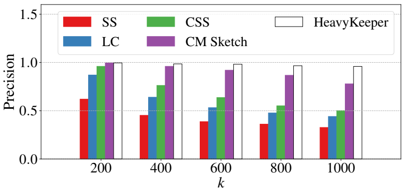


Figure 7: Precision vs. *k* (Campus).

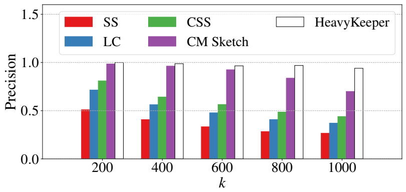


Figure 8: Precision vs. *k* (CAIDA).

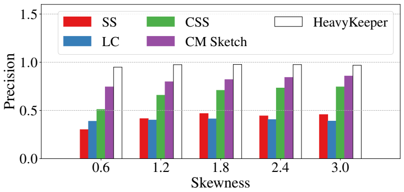


Figure 9: Precision vs. skewness (Synthetic).

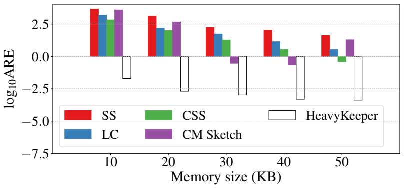


Figure 10: ARE vs. memory size (Campus).

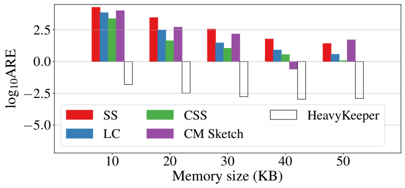


Figure 11: ARE vs. memory size (CAIDA).

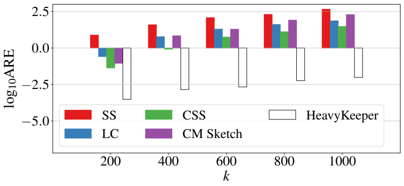


Figure 12: ARE vs. *k* (Campus).

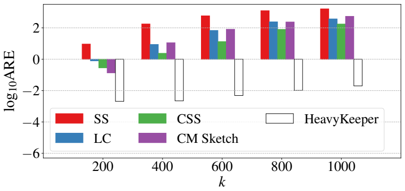


Figure 13: ARE vs. *k* (CAIDA).

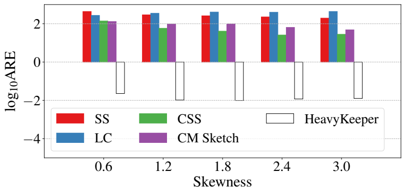


Figure 14: ARE vs. skewness (Synthetic).

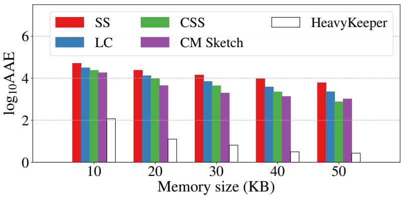


Figure 15: AAE vs. memory size (Campus).

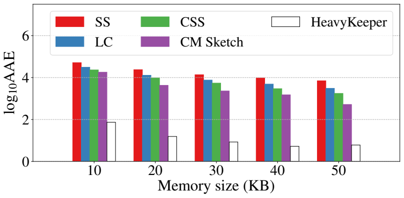


Figure 16: AAE vs. memory size (CAIDA).

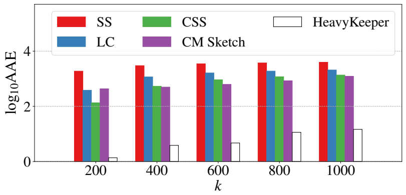


Figure 17: AAE vs. *k* (Campus).

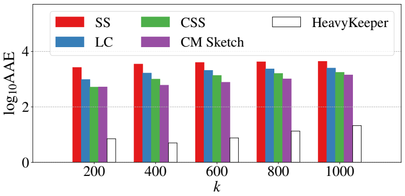


Figure 18: AAE vs. *k* (CAIDA).

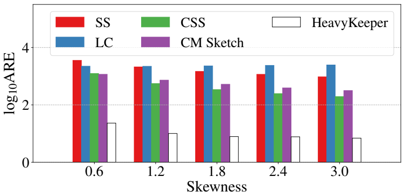


Figure 19: AAE vs. skewness (Synthetic).

sketch is respectively 10%, 11%, 19%, and 41%, while the one of HeavyKeeper is 82%. Furthermore, we find that the precision of HeavyKeeper reaches 100% for a memory size of 30KB, while the corresponding precision of Space-Saving, Lossy counting, CSS, and CM sketch is 27%, 39%, 49%, and 93%. This implies that HeavyKeeper has indeed much better precision than the other three algorithms. We find that Lossy counting is more accurate than Space-Saving. However, as will be mentioned later, Lossy counting is much slower than the other algorithms. For the CAIDA dataset (see Figure 6), we find that the precision of HeavyKeeper reaches 99.99% when memory size is larger than 20KB, while for Space-Saving, Lossy counting, CSS, and CM sketch, precision is respectively 18%, 33%, 34%, and 89% when memory size is 50KB.

Precision vs. *k*: As shown in Figure 7, for the campus dataset, as *k* becomes larger, the precision of HeavyKeeper stays high, while it degrades for other algorithms. For the campus dataset, as *k* becomes larger, the precision of HeavyKeeper is always higher than 95.9%, while that of Space-Saving, Lossy counting, CSS, and CM sketch reaches 32.7%, 44.1%, 50.1%, and 77.9% respectively when $k = 1000$. This happens for two main reasons: 1) larger *k* requires larger memory usage to store information about more flows; 2) as *k* increases, the difference of flow sizes among flows becomes smaller, so it is easy to mistake other flows for top-*k* flows. For the CAIDA dataset (Figure 8), we find that the precision of HeavyKeeper is always above 94%, while for Space-Saving, Lossy counting, CSS, and CM sketch, it is 26.6%, 37.1%, 44%, and 70% respectively when $k = 1000$.

Precision vs. skewness: As shown in Figure 9, the precision of HeavyKeeper reaches 99.99%. For all considered values of skewness, the precision of HeavyKeeper does not go below 94.9%, while the highest precision for Space-Saving, Lossy counting, CSS, and CM sketch is 46.8%, 41.3%, 74.5%, and 85.7%, respectively.

5.4 Experiments on AAE and ARE

In this section, we focus on the ARE and the AAE of the estimated frequency of reported top-*k* flows. We also conduct experiments with varying memory size, *k*, and skewness. The parameter settings are the same as in Section 5.3.

ARE vs. memory size: As shown in Figure 10, for the campus dataset, we find that the ARE of HeavyKeeper is smaller than 0.01 when memory size is larger than 20KB, while for Space-Saving, Lossy counting, CSS, and CM sketch, it is larger than 100. Furthermore, we find that the ARE of HeavyKeeper is between 100158 and 648291 times smaller than the one of Space-Saving, between 8450 and 78209 times smaller than the one of Lossy counting, between 945 and 49561 times smaller than the one of CSS, and between 279 and 226986 times smaller than the one of CM sketch. For the CAIDA dataset (see Figure 11), we find that the ARE of HeavyKeeper is between 21119 and 1190365 times smaller than the one of Space-Saving, between 2955 and 456275 times smaller than the one of Lossy counting, between 950 and 154047 times smaller than the one of CSS, and between 238 and 656145 times smaller than the one of CM sketch.

ARE vs. *k*: As shown in Figure 12, for the campus dataset, we find that the ARE of HeavyKeeper is between 25579 and 56791 times smaller than the one of Space-Saving, between 852 and 9312 times smaller than the one of Lossy counting, between 142 and 3132 times smaller than the one of CSS, and between 293 and 20370 times smaller than the of of CM sketch. For the CAIDA dataset (see Figure 13), we find that the ARE of HeavyKeeper is between 4506 and 121912 times smaller than the one of Space-Saving, between 383 and 23666 times smaller than the one of Lossy counting, between 137 and 8816 times smaller than the one of CSS, and between 66 and 27290 times smaller than the one of CM sketch.

ARE vs. skewness: As shown in Figure 14, for all considered values of skewness, we find that the ARE of HeavyKeeper is between 15566 and 27829 times smaller than that of Space-Saving, between 11915 and 41575 times smaller than that of Lossy counting, between 2174 and 6099 times smaller than that of CSS, and between 3819 and 10080 times smaller than that of CM sketch.

AAE vs. memory size: As shown in Figure 15, for the campus dataset, we find that the AAE of HeavyKeeper is between 433 and 3013 times smaller than that of Space-Saving, between 267 and 1221 times smaller than that of Lossy counting, between 200 and 758 times smaller than that of CSS, and between 155 and 428 times smaller than that of CM sketch. When memory size is 50KB, the AAE of HeavyKeeper is only 2.73, confirming that the estimated flow sizes of almost all reported flows are accurate. For the CAIDA dataset (see Figure 16), we find that the AAE of HeavyKeeper is between 697 and 1810 times smaller than that of Space-Saving, between 421 and 928 times smaller than that Lossy counting, between 289 and 647 times smaller than the one of CSS, and between 86 and 284 times smaller than that of CM sketch.

AAE vs. *k*: As shown in Figure 17, for the campus dataset, we find that the AAE of HeavyKeeper is between 271 and 1382 times smaller than that of Space-Saving, between 142 and 346 times smaller than that of Lossy counting, between 93 and 196 times smaller than that of CSS, and between 74 and 318 times smaller than that of CM sketch. For CAIDA dataset (see Figure 18), we find that the AAE of HeavyKeeper is between 206 and 694 times smaller than that of Space-Saving, between 118 and 329 times smaller than that of Lossy counting, between 73 and 199 times smaller than that of CSS, and between 67 and 121 times smaller than that of CM sketch.

AAE vs. skewness: From Figure 19, we find that the AAE of HeavyKeeper is between 137 and 209 times smaller than that of Space-Saving, between 96 and 355 times smaller than that of Lossy counting, between 28 and 55 times smaller than that of CSS, and between 45 and 73 times smaller than that of CM sketch.

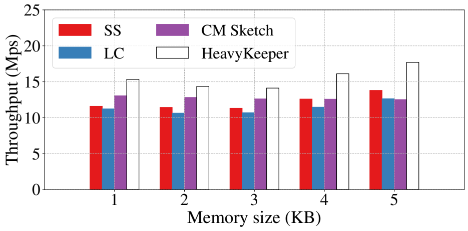


Figure 20: Throughput vs. memory size.

5.5 Experiments on Throughput

We now turn to the throughput of the algorithms. We only report results for the campus dataset due to space limitations. We set $k = 100$, and vary memory size from 10KB to 50KB. Here we use CAIDA datasets.

Throughput vs. memory size: As shown in Figure 20, we find that the throughput of HeavyKeeper is always higher than other algorithms. Indeed, the average throughput of HeavyKeeper is 15.52Mps, while it is 12.15Mps, 11.34Mps, and 12.72Mps for Space-Saving, Lossy counting, and CM sketch. These results show that HeavyKeeper not only is more accurate than previous work, but also achieves higher throughput as well.

6 Open vSwitch Deployment

In this section, we implement our HeavyKeeper algorithm on a software switch platform: Open vSwitch (OVS). We first present details of our implementation, and then present experimental results to show the performance of our algorithm running on Open vSwitch.

6.1 OVS Implementation

The OVS implementation of our HeavyKeeper algorithm consists of three components: 1) the modified OVS datapath, 2) the shared memory buffering flow IDs, and 3) the user-space program of HeavyKeeper processing flow IDs. For each incoming packet, it will be first inserted into the OVS datapath for forwarding. Besides, we modify the source codes of OVS datapath to parse the flow ID of the incoming packet, and then insert its flow ID into the shared memory (the shared memory is created initially). Finally, the user-space program will read the flow IDs from the shared memory, and process them as incoming packets.

In order to improve the performance of OVS, we integrate OVS with DPDK (Data Plane Development Kit). DPDK implements the datapath entirely in the user-space, and thus it eliminates the overhead of a context switch and memory copies between user-space and kernel-space.

6.2 OVS Evaluation

To evaluate the performance of HeavyKeeper implemented in OVS, we conduct experiments to evaluate the throughput of HeavyKeeper and other algorithms. Besides, we also show the throughput of OVS without using any algorithm to show the impact of algorithms. We set the memory size to 50KB.

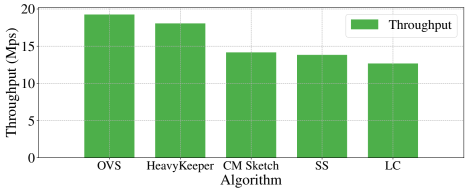


Figure 21: Throughput on OVS platform.

As shown in Figure 21, the throughput of HeavyKeeper is near the original throughput of OVS. Specifically, the throughput of the original OVS is 19.22Mps, and that of HeavyKeeper is 18.03Mps. However, the throughput of CM sketch, Space-Saving, and Lossy Counting is 14.14Mps, 13.80Mps, and 12.64Mps, respectively. The results show that our HeavyKeeper algorithm has little impact to the performance of OVS, while other algorithms decrease the throughput significantly.

7 Conclusion

Finding the top-k elephant flows is a critical task for network traffic measurement. As the line rate increases, it is more and more challenging to design an accurate algorithm that achieves fast and constant speed. Existing algorithms for finding top-k flows cannot achieve high precision when traffic speed is high and memory usage is small, because they do not handle massive mouse flows effectively. In this paper, we propose a novel data structure, called HeavyKeeper, which achieves a much higher precision on top-k queries and a much lower error rate on flow size estimation, compared to previous algorithms. The key idea of HeavyKeeper is that it intelligently omits mouse flows, and focuses on recording the information of elephant flows by using the exponential-weakening decay strategy. Our evaluation confirms that HeavyKeeper achieves 99.99% precision for finding the top-k elephant flows, while also achieving a reduction in the error rate of the estimated flow size by about 3 orders of magnitude compared to the state-of-the-art algorithms.

References

[1] Anirudh Sivaraman, Suvinay Subramanian, and et al. Programmable packet scheduling at line rate. In *Proc. ACM SIGCOMM*, 2016.

[2] Anja Feldmann, Albert Greenberg, and et al. Deriving traffic demands for operational ip networks: Methodology and experience. In *Proc. ACM SIGCOMM*, 2000.

[3] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *Proc. ACM IMC*, 2004.

[4] Ori Rottenstreich and János Tapolcai. Optimal rule caching and lossy compression for longest prefix matching. *IEEE/ACM Transactions on Networking*, 25(2):864–878, 2017.

[5] Katsiaryna Mirylenka, Graham Cormode, Themis Palpanas, and Divesh Srivastava. Conditional heavy hitters: detecting interesting correlations in data streams. *The VLDB Journal*, 24(3):395–414, 2015.

[6] Joong Hyuk Chang and Won Suk Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proc. ACM SIGKDD*, pages 487–492. ACM, 2003.

[7] Yin-Ling Cheung and Ada Wai-Chee Fu. Mining frequent itemsets without support threshold: with and without item constraints. *IEEE TKDE*, 16(9):1052–1069, 2004.

[8] Gerard Salton and Michael J McGill. Introduction to modern information retrieval. 1986.

[9] Mohamed A Soliman, Ihab F Ilyas, and Kevin Chen-Chuan Chang. Top-k query processing in uncertain databases. In *Proc. IEEE ICDE*, pages 896–905, 2007.

[10] Yu Zhang, BinXing Fang, and YongZheng Zhang. Identifying heavy hitters in high-speed network monitoring. *Science China Information Sciences*, 53(3):659–676, 2010.

[11] Elisa Bertino. Introduction to data security and privacy. *Data Science and Engineering*, 1(3):125–126, 2016.

[12] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proc. SIGMOD 2016*.

[13] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

[14] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[15] Cristian Estan and George Varghese. *New directions in traffic measurement and accounting*, volume 32. ACM, 2002.

[16] Yin Zhang, Matthew Roughan, Walter Willinger, and Lili Qiu. Spatio-temporal compressive sensing and internet traffic matrices. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 267–278. ACM, 2009.

[17] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[18] Graham Cormode, Balachander Krishnamurthy, and Walter Willinger. A manifesto for modeling and measurement in social media. *First Monday*, 15(9), 2010.

[19] Dave Maltz. Unraveling the complexity of network management. 2009.

[20] Ilker Nadi Bozkurt, Yilun Zhou, Theophilus Benson, Bilal Anwer, Dave Levin, Nick Feamster, Aditya Akella, Balakrishnan Chandrasekaran, Cheng Huang, Bruce Maggs, et al. Dynamic prioritization of traffic in home networks. 2015.

[21] Zhetao Li, Fu Xiao, Shiguo Wang, Tingrui Pei, and Jie Li. Achievable rate maximization for cognitive hybrid satellite-terrestrial networks with af-relays. *IEEE Journal on Selected Areas in Communications*, 36(2):304–313, 2018.

[22] Muhammad Habib ur Rehman, Chee Sun Liew, Assad Abbas, Prem Prakash Jayaraman, Teh Ying Wah, and Samee U Khan. Big data reduction methods: a survey. *Data Science and Engineering*, 1(4):265–284, 2016.

[23] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *Proc. IEEE INFOCOM*, 2016.

[24] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. Springer ICDT 2005*.

[25] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Automata, languages and programming*, pages 784–784, 2002.

[26] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proc. VLDB*, pages 346–357, 2002.

[27] Zhetao Li, Baoming Chang, Shiguo Wang, Anfeng Liu, Fanzi Zeng, and Guangming Luo. Dynamic compressive wide-band spectrum sensing based on channel energy reconstruction in cognitive internet of things. *IEEE Transactions on Industrial Informatics*, 2018.

[28] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI*, pages 311–324, 2016.

[29] Wang Feng and Hamdi Mounir. Matching the speed gap between sram and dram. In *Proc. IEEE HSPR*, pages 104–109, 2008.

[30] Erik Demaine, Alejandro López-Ortiz, and J Munro. Frequency estimation of internet packet streams with limited space. *AlgorithmsESA 2002*, pages 11–20, 2002.

[31] The source codes of heavykeeper and other related algorithms. `https://github.com/papergitkeeper/heavy-keeper-project`.

[32] Gil Einziger and Roy Friedman. Counting with tinytable: Every bit counts! In *Proc. ICDCN 2016*.

[33] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proc. ACM IMC 2004*.

[34] The caida anonymized internet traces 2016. `http://www.caida.org/data/overview/`.

[35] David MW Powers. Applications and explanations of zipf's law. In *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*, pages 151–160. Association for Computational Linguistics, 1998.

[36] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 2004.

SAND: Towards High-Performance Serverless Computing

Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein
Klaus Satzke, Andre Beck, Paarijaat Aditya, Volker Hilt
Nokia Bell Labs

Abstract

Serverless computing has emerged as a new cloud computing paradigm, where an application consists of individual functions that can be separately managed and executed. However, existing serverless platforms normally isolate and execute functions in separate containers, and do not exploit the interactions among functions for performance. These practices lead to high startup delays for function executions and inefficient resource usage.

This paper presents SAND, a new serverless computing system that provides lower latency, better resource efficiency and more elasticity than existing serverless platforms. To achieve these properties, SAND introduces two key techniques: 1) application-level sandboxing, and 2) a hierarchical message bus. We have implemented and deployed a complete SAND system. Our results show that SAND outperforms the state-of-the-art serverless platforms significantly. For example, in a commonly-used image processing application, SAND achieves a 43% speedup compared to Apache OpenWhisk.

1 Introduction

Serverless computing is emerging as a key paradigm in cloud computing. In serverless computing, the unit of computation is a function. When a service request is received, the serverless platform allocates an ephemeral execution environment for the associated function to handle the request. This model, also known as Function-as-a-Service (FaaS), shifts the responsibilities of dynamically managing cloud resources to the provider, allowing the developers to focus only on their application logic. It also creates an opportunity for the cloud providers to improve the efficiency of their infrastructure resources.

The serverless computing paradigm has already created a significant interest in industry and academia. There have been a number of commercial serverless offerings (e.g., Amazon Lambda [1], IBM Cloud Functions [5], Microsoft Azure Functions [11], and Google Cloud Functions [15]), as well as several new proposals (e.g., OpenLambda [24] and OpenFaaS [19]).

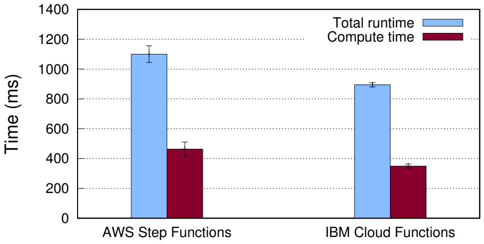


Figure 1: Total runtime and compute time of executing an image processing pipeline with four functions on existing commercial serverless platforms. Results show the mean values with 95% confidence interval over 10 runs, after discarding the initial (cold) execution.

Although existing serverless platforms work well for simple applications, they are not well-suited for more complex services due to their overheads, especially when the application logic follows an execution path spanning multiple functions. Consider an image processing pipeline that executes four consecutive functions [51]: extract image metadata, verify and transform it to a specific format, tag objects via image recognition, and produce a thumbnail. We ran this pipeline using AWS Step Functions [10] and IBM Cloud Functions with Action Sequences [29], both of which provide a method to connect multiple functions into a single service.[1] On these platforms, we found that the total runtime is significantly more than the actual time required for function executions (see Figure 1), indicating the overheads of executing such connected functions. As a result of these overheads, the use and adoption of serverless computing by a broader range of applications is severely limited.

We observe two issues that contribute to these over-

[1] As of this writing, other major serverless providers, e.g., Microsoft and Google, do not support Python, which was used for this pipeline.

heads and diminish the benefits of serverless computing. Our first observation is that most existing serverless platforms execute each application function within a separate container instance. This decision leads to two common practices, each with a drawback. First, one can start a new, 'cold' container to execute an associated function every time a new request is received, and terminate the container when the execution ends. This approach inevitably incurs long startup latency for each request. The second practice is to keep a launched container 'warm' (i.e., running idle) for some time to handle future requests. While reducing the startup latency for processing, this practice comes at a cost of occupying system resources unnecessarily for the idle period (i.e., resource inefficiency).

Our second observation is that existing serverless platforms do not appear to consider interactions among functions, such as those in a sequence or workflow, to reduce latency. These platforms often execute functions wherever required resources are available. As such, external requests (e.g., user requests calling the first function in a workflow) are treated the same as internal requests (e.g., functions initiating other functions during the workflow execution), and load is balanced over all available resources. This approach causes function interactions to pass through the full end-to-end function call path, incurring extra latency. For example, in Apache OpenWhisk [5] (i.e., the base system of IBM Cloud Functions [29]), even for functions that belong to the same workflow defined by the Action Sequences, all function calls pass through the controller. We also observe that, for functions belonging to the same state machine defined by AWS Step Functions [10], the latencies between functions executing on the same host are similar to the latencies between functions running on different hosts.

In this paper, we design and prototype a novel, high-performance serverless platform, SAND. Specifically, we propose two mechanisms to accomplish both low latency and high resource efficiency. First, we design a *fine-grained application sandboxing mechanism* for serverless computing. The key idea is to have two levels of isolation: 1) isolation between different applications, and 2) isolation between functions of the same application. This distinction enables SAND to quickly allocate (and deallocate) resources, leading to low startup latency for functions and efficient usage of cloud resources. We argue that stronger mechanisms (e.g., containers) are needed only for the isolation among applications, and weaker mechanisms (e.g., processes and light-weight contexts [37]) are well-suited for the isolation among functions within the same application.

Second, we design a *hierarchical message queuing and storage mechanism* to leverage locality for the interacting functions of the same application. Specifically, SAND orchestrates function executions of the same application as local as possible. We develop a local message bus on each host to create shortcuts to enable fast message passing between interacting functions, so that functions executing in a sequence can start almost instantly. In addition, for reliability, we deploy a global message bus that serves as a backup of locally produced and consumed messages in case of failures. The same hierarchy is also applied for our storage subsystem in case functions of the same application need to share data.

With these two mechanisms, SAND achieves low function startup and interaction latencies, as well as high resource efficiency. Low latencies are crucial for serverless computing and play a key role in broadening its use. Otherwise, application developers would merge functions to avoid the latency penalty, making the applications less modular and losing the benefits of serverless computing. In addition, the cloud industry is increasingly interested in moving infrastructure to the network edge to further reduce network latency to users [18, 26, 36, 50, 52]. This move requires high resource efficiency because the edge data centers typically have fewer resources than the core.

We implemented and deployed a complete SAND system. Our evaluation shows that SAND outperforms state-of-the-art serverless platforms such as Apache OpenWhisk [5] by 8.3× in reducing the interaction latency between (empty) functions and much more between typical functions. For example, in a commonly-used image processing application, these latencies are reduced by 22×, leading to a 43% speedup in total runtime. In addition, SAND can allocate and deallocate system resources for function executions much more efficiently than existing serverless platforms. Our evaluation shows that SAND improves resource efficiency between 3.3× and two orders of magnitude compared to the state-of-the-art.

2 Background

In this section, we give an overview of existing serverless platforms, their practices, and the implications of these practices. We summarize how these platforms deploy and invoke functions in parallel as well as in sequence. Our observations are based on open-source projects, but we think they still reflect many characteristics of commercial platforms. For example, the open-source Apache OpenWhisk is used in IBM Cloud Functions [29]. In addition, we augment these observations with our experiences using these platforms and with publicly available information from the commercial providers.

2.1 Function Deployment

In serverless computing, the cloud operator takes the responsibility of managing servers and system resources to run the functions supplied by developers. To our knowledge, the majority of serverless platforms use containers and map each function into its own container to achieve this goal. This mapping enables the function code to be portable, so that the operator can execute the function wherever there are enough resources in the infrastructure without worrying about compatibility. Containers also provide virtually isolated environments with namespaces separating operating system resources (e.g., processes, filesystem, networking), and can isolate most of the faulty or malicious code execution. Serverless platforms that employ such a mapping include commercial platforms (e.g., AWS Lambda, Microsoft Azure Functions, Google Cloud Functions, and IBM Cloud Functions) as well as open-source platforms (e.g., Apache OpenWhisk, OpenLambda [24], Greengrass [7], and OpenFaaS [19]).

Note that there are also other platforms that employ NodeJS [45] to run functions written in JavaScript [25, 49, 54, 58]. These platforms offer alternatives for serverless computing, but are not as widely used as the container-based platforms. For this reason, hereafter we describe in detail serverless platforms that employ containers and use them in our evaluation.

2.2 Function Call

The most straightforward approach to handle an incoming request is to start a new container with the associated function code and then execute it. This approach, known as 'cold' start, requires the initialization of the container with the necessary libraries, which can incur undesired startup latency to the function execution. For example, AWS Lambda has been known to have delays of up to a few seconds for 'cold' function calls [8]. Similarly, Google has reported a median startup latency of 25 seconds on its internal container platform [55], 80% of which are attributed to library installation. Lazy loading of libraries can reduce this startup latency, but it can still be on the order of a few seconds [23].

To improve startup latency, a common practice is to reuse launched containers by keeping them 'warm' for a period of time. The first call to a function is still 'cold', but subsequent calls to this function can be served by the 'warm' container to avoid undesired startup latency. To also reduce the latency of the first function call, Apache OpenWhisk can launch containers even before a request arrives via the 'pre-warming' technique [5].

The above 'warm' container practice, however, unnecessarily occupies resources with idling containers. Note that this practice also relaxes the original isolation guarantee provided by the containers, because different requests may be handled inside the same container albeit sequentially (i.e., one execution at a time).

2.3 Function Concurrency

Another aspect in which various serverless platforms can differ is how they handle concurrent requests. Apache OpenWhisk and commercial platforms such as AWS Lambda [1], Google Cloud Functions and Microsoft Azure Functions allow only one execution at a time in a container for performance isolation. As a result, concurrent requests will either be handled in their individual containers and experience undesired startup latencies for each container, or the requests will be queued for a 'warm' container to become available and experience queuing delays. In contrast, OpenFaaS [19] and OpenLambda [24] allow concurrent executions of the same function in a single container.

2.4 Function Chaining

Application logic often consists of sequences of multiple functions. Some existing serverless platforms support the execution of function sequences (e.g., IBM Action Sequences [29], AWS Step Functions [10]). In a function sequence, the events that trigger function executions can be categorized as external (e.g., a user request calling a function sequence) and internal (e.g., a function initiating other functions during the workflow execution). Existing serverless platforms normally treat these events the same, such that each event traverses the full end-to-end function call path (e.g., event passing via a unified message bus or controller), incurring undesired latencies.

3 SAND Key Ideas and Building Blocks

This section describes the key ideas and building blocks of SAND. We first present the design of our application sandboxing mechanism (§3.1) that enables SAND to be resource-efficient and elastic as well as to achieve low-latency function interactions. Then, we describe our hierarchical message queuing mechanism (§3.2) that further reduces the function interaction latencies.

3.1 Application Sandboxing

The key idea in our sandboxing design is that we need two levels of fault isolation: 1) isolation between different applications, and 2) isolation between functions of the same application. Our reasoning is that different applications require strong isolation from each other. On the other hand, functions of the same application may not need such a strong isolation, allowing us to improve the performance of the application. Note that some existing serverless platforms reuse a 'warm' container to execute calls to the same function, making a similar trade-off to improve the performance of a single function.

To provide a two-level fault isolation, one can choose from a variety of technologies, such as virtual machines (VMs), LightVMs [40], containers [21, 53], unikernels [38, 39], processes, light-weight contexts [37] and threads. This choice will impact not only performance but also the dynamic nature of applications and functions as well as the maintenance effort by cloud operators. We discuss these implications in §9.

In SAND, we specifically separate applications from each other via containers, such that each application runs in its own container. The functions that compose an application run in the same container but as separate processes. Upon receiving an incoming request, SAND forks a new process in the container to execute the associated function, such that each request is handled in a separate process. For example, Figure 2 shows that the two functions f_1 and f_2 of the same application are run in the same application sandbox on a host, but different applications are separated.

Our application sandboxing mechanism has three significant advantages. First, triggering a function execution by forking a process within a container incurs short startup latency, especially compared to launching a separate container per request or function execution — up to three orders of magnitude speedup (§6.1). Second, the libraries shared by multiple functions of an application need to be loaded into the container only once. Third, the memory footprint of an application container increases in small increments with each incoming request and decreases when the request has been processed, with the resources allocated for a single function execution being released immediately (i.e., when the process terminates). As a result, the cloud operator can achieve substantially better resource efficiency and has more flexibility to divert resources not only among a single application's functions but also among different applications (i.e., no explicit pre-allocation). This effect becomes even more critical in emerging edge computing scenarios where cloud infrastructure moves towards the network edge that has only limited resources.

3.2 Hierarchical Message Queuing

Serverless platforms normally deploy a unified message bus system to provide scalable and reliable event dispatching and load balancing. Such a mechanism works well in scenarios where individual functions are triggered via (external) user requests. However, it can cause unnecessary latencies when multiple functions interact with each other, such that one function's output is the input to another function. For example, even if two functions of an application are to be executed in a sequence and they reside on the same host, the trigger message between the two functions still has to be published to the unified message bus, only to be delivered back to the same host.

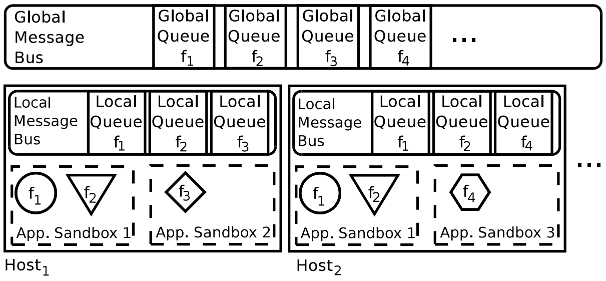


Figure 2: SAND's key building blocks: application sandboxing and hierarchical message queuing.

To address this problem, we design a *hierarchical* message bus for SAND. Our basic idea is to create shortcuts for functions that interact with each other (e.g., functions of the same application). We describe the hierarchical message bus and its coordination with two levels.[2]

In a two-level hierarchy, there is a *global* message bus that is distributed across hosts and a *local* message bus on every host (Figure 2). The global message bus serves two purposes. First, it delivers event messages to functions across different hosts, for example, when a single host does not have enough resources to execute all desired functions or an application benefits from executing its functions across multiple hosts (e.g., application sandbox 1 in Figure 2). Second, the global message bus also serves as a backup of local message buses for reliability.

The local message bus on each host is used to deliver event messages from one function to another if both functions are running on the same host. As a result, the interacting functions (e.g., an execution path of an application spanning multiple functions, similar to the application sandbox 1 in Figure 2) can benefit from reduced latency because accessing the local message bus is much faster than accessing the global message bus (§6.2).

The local message bus is first-in-first-out, preserving the order of messages from one function to another. For global message bus, the order depends on the load balancing scheme: if a shared identifier of messages (e.g., key) is used, the message order will also be preserved.

Coordination. To ensure that an event message does not get processed at the same time on multiple hosts, the local and global message buses coordinate: a backup of the locally produced event message is published to the global message bus with a *condition flag*. This flag indicates that the locally produced event message is being processed on the current host and should not be delivered to another host for processing. After publishing the backup message, the current host tracks the progress of the forked process that is handling the event message and

[2] This hierarchy can be extended to more than two levels in a large network; we omit the description due to space limit.

updates its flag value in the global message bus accordingly (i.e., ‘processing’ or ‘finished’). If the host fails after the backup message is published to the global message bus but before the message has been fully processed, another host takes over the processing after a timeout.

This coordination procedure is similar to write-ahead-logging in databases [57], whereby a locally produced event message would be first published to the global message bus before the local message bus. While guaranteeing that there are no lost event messages due to host failures, this ‘global-then-local’ publication order can add additional latency to the start of the next (locally available) function in a sequence. In SAND, we relax this order, and publish event messages to the global message bus asynchronously with the publication to the local message bus in parallel. In serverless computing, functions are expected to, and usually, finish execution fast [9,12]. In case of a failure, SAND can reproduce the lost event messages by re-executing the functions coming after the last (backup) event message seen in the global message bus. Note that, in SAND, the output of a function execution becomes available to other functions at the end of the function execution (§4.1). As such, the coordination and recovery procedures work for outputs that are contained within SAND. SAND does not guarantee the recovery of functions that make externally-visible side effects *during* their executions, such as updates to external databases.

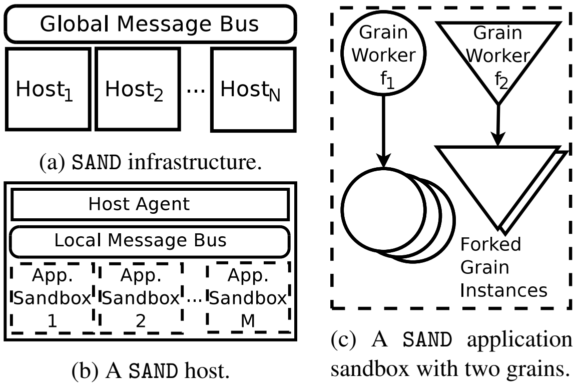
4 SAND System Design

This section presents the detailed design of SAND utilizing the aforementioned key ideas and building blocks. We also illustrate how an example application runs on SAND, and describe some additional system components.

4.1 System Components

The SAND system contains a number of hosts, which can exchange event messages via a global message bus (Figure 3a). Figure 3b shows the system components on a single host. Here, we describe these components.

Application, Grain, and Workflow. In SAND, a function of an application is called a *grain*. An *application* consists of one or multiple grains, as well as the *workflows* that define the interactions among these grains. The interaction between grains can be *static* where the grains are chained (e.g., Grain_2's execution always follows Grain_1's execution), or *dynamic* where the execution path is determined during the execution (e.g., the execution of Grain_2 and/or Grain_3 may follow Grain_1's execution, according to Grain_1's output). The grain code and workflows are supplied by the application developer. A grain can be used by multiple applications by copying it to the respective application sandboxes.



(a) SAND infrastructure.

(b) A SAND host.

(c) A SAND application sandbox with two grains.

Figure 3: High-level architecture of SAND.

Sandbox, Grain Worker, and Grain Instance. An application can run on multiple hosts. On a given host, the application has its own container called *sandbox*. The set of grains hosted in each sandbox can vary across hosts as determined by the application developer[3], but usually includes all grains of the application.

When a sandbox hosts a specific grain, it runs a dedicated OS process called *grain worker* for this grain. The grain worker loads the associated grain code and its libraries, subscribes to the grain's queue in the host's local message bus, and waits for event messages.

Upon receiving an associated event message, the grain worker forks itself to create a *grain instance* that handles the event message (Figure 3c). This mechanism provides three significant advantages for SAND. First, forking grain instances from the grain worker is quite fast and lightweight. Second, it utilizes OS mechanisms that allow the sharing of initialized code (e.g., loaded libraries), thus reducing the application's memory footprint. Third, the OS automatically reclaims the resources assigned to a grain instance upon its termination. Altogether, by exploiting the process forking, SAND becomes fast and efficient in allocating and deallocating resources for grain executions. As a result, SAND can easily handle load variations and spikes to multiplex multiple applications (and their grains) elastically even on a single host.

When a grain instance finishes handling an event message, it produces the output that includes zero or more event messages. Each such message is handled by the next grain (or grains), as defined in the workflow of the application. Specifically, if the next grain is on the same host, the previous grain instance directly publishes the output event message into the local message queue that is subscribed to by the next grain worker, which then forks a grain instance to handle this event message. In parallel, a backup of this message is asynchronously published to the global message bus.

[3] Or automatically by SAND via strategies or heuristics (e.g., a sandbox on each host should not run more than a certain number of grains).

Local and Global Message Buses. A *local message bus* runs on each host, and serves as a shortcut for local function interactions. Specifically, in the local message bus, a separate message queue is created for each grain running on this host (Figure 2). The local message bus accepts, stores and delivers event messages to the corresponding grain worker when it polls its associated queue.

On the other hand, the *global message bus* is a distributed message queuing system that runs across the cloud infrastructure. The global message bus acts as a backup for locally produced and consumed event messages by the hosts, as well as delivers event messages to the appropriate remote hosts if needed. Specifically, in the global message bus, there is an individual message queue associated with each grain hosted in the entire infrastructure (see Figure 2). Each such message queue is partitioned to increase parallelism, such that each partition can be assigned to a different host running the associated grain. For example, the widely-used distributed message bus Apache Kafka [3] follows this approach.

Each host synchronizes its progress on the consumption of the event messages from their respective partitions with the global message bus. In case of a failure, the failed host's partitions are reassigned to other hosts, which then continue consuming the event messages from the last synchronization point.

Host Agent. Each host in the infrastructure runs a special program called *host agent*. The host agent is responsible for the coordination between local and global message buses, as well as launching sandboxes for applications and spawning grain workers associated with the grains running on this host. The host agent subscribes to the message queues in the global message bus for all the grains the host is currently running. In addition, the host agent tracks the progress of the grain instances that are handling event messages, and synchronizes it with the host's partitions in the global message bus.

4.2 Workflow Example

The SAND system can be best illustrated with a simple workflow example demonstrating how a user request to a SAND application is handled. Suppose the application consists of two grains, Grain_1 and Grain_2. Host_x is running this application with the respective grain workers, GW_1 and GW_2. The global message bus has an individual message queue associated with each grain, GQ_1 and GQ_2. In addition, there is an individual partition (from the associated message queue in the global message bus) assigned to each of the two grain workers on Host_x, namely $\text{GQ}_{1,1}$ and $\text{GQ}_{2,1}$, respectively.

Assume there is a user request for Grain_1 (Step 0 in Figure 4), and the global message bus puts this event message into the partition $\text{GQ}_{1,1}$ within the global message queue GQ_1, according to a load balancing strategy. As a result, the host agent on Host_x can retrieve this event message (Step 1) and publish it into the local queue LQ_1 (associated with Grain_1) in Host_x's local message bus (Step 2). The grain worker GW_1, which is responsible for Grain_1 and subscribed to LQ_1, retrieves the recently added event message (Step 3) and forks a new grain instance (i.e., a process) to handle the message (Step 4).

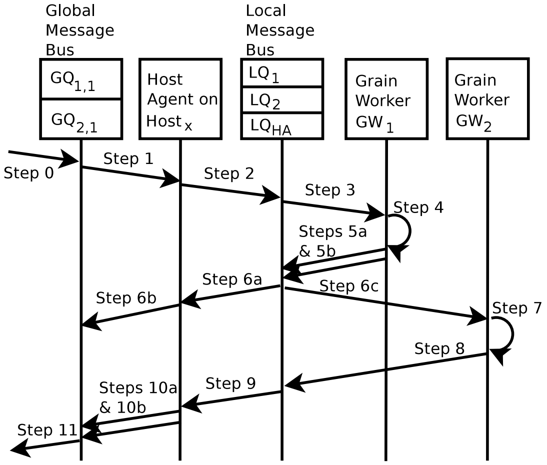


Figure 4: Handling of a user request to a simple application that consists of two grains in a workflow.

Assume Grain_1's grain instance produces a new event message for the next grain in the workflow, Grain_2. The grain instance publishes this event message directly to Grain_2's associated local queue LQ_2 (Step 5a), because it knows that Grain_2 is locally running. A copy of the event message is also published to the local queue LQ_{HA} for the host agent on Host_x (Step 5b). The host agent retrieves the message (Step 6a) and publishes it as a backup to the assigned partition $\text{GQ}_{2,1}$ in Grain_2's associated global queue with a condition flag 'processing' (Step 6b).

Meanwhile, the grain worker GW_2 for Grain_2 retrieves the event message from the local queue LQ_2 in the local message bus on Host_x (Step 6c). GW_2 forks a new grain instance, which processes the event message and terminates after execution (Step 7). In our example, GW_2 produces a new event message to the local queue of the host agent LQ_{HA} (Step 8), because Grain_2 is the last grain in the application's workflow and there are no other locally running grains to handle it. The host agent retrieves the new event message (Step 9) and directly publishes it to the global message bus (Step 10a). In addition, the finish of the grain instance of Grain_2 causes the host agent to update the condition flag of the locally produced event message that triggered Grain_2 with a value 'finished' to indicate that it has been successfully processed (Step 10b). The response is then sent to the user (Step 11).

4.2.1 Handling Host Failures

In the previous description, all hosts are alive during the course of the workflow. Suppose the processing of the event message for Grain_2 (Step 7 in Figure 4) failed due to the failure of Host_x. When the global message bus detects Host_x's failure, it will reassign Host_x's associated partitions (i.e., $\text{GQ}_{1,1}$ and $\text{GQ}_{2,1}$). Suppose there is another host Host_y taking over these two partitions. In our example, only Grain_2's grain instances were triggered via the locally produced event messages, meaning that the condition flags were published to $\text{GQ}_{2,1}$ (i.e., Host_x's partition in Grain_2's global message queue).

When Host_y starts the recovery process, it retrieves all event messages in $\text{GQ}_{2,1}$ (and also $\text{GQ}_{1,1}$). For each message, Host_y's host agent checks its condition flag. If the flag indicates that an event message has been processed successfully (i.e., 'finished'), this message is skipped because Host_x failed after processing this event message. If the flag indicates that the processing of the event message has just started (i.e., 'processing'), Host_y processes this event message following the steps in Figure 4.

It is possible that Host_y fails during the recovery process. To avoid the loss of event messages, Host_y continuously synchronizes its progress on the consumed messages from the reassigned partitions with the global message bus. It does not retrieve any new messages from the partitions until all messages of the failed host have been processed successfully. As a result, each host replacing a failed host deals with smaller reassigned partitions.

4.3 Additional System Components

Here, we briefly describe a few additional system components that complete SAND.

Frontend Server. The *frontend server* is the interface for developers to deploy their applications as well as to manage grains and workflows. It acts as the entry point to any application on SAND. For scalability, multiple frontend servers can run behind a standard load balancer.

Local and Global Data Layers. Grains can share data by passing a reference in an event message instead of passing the data itself. The *local data layer* runs on each host similar to the local message bus, and enables fast access to the data that local grains want to share among themselves via an in-memory key-value store. The *global data layer* is a distributed data storage running across the cloud infrastructure similar to the global message bus. The coordination between the local and global data layers is similar to the coordination between the local and global message buses (§3.2). Each application can only access its own data in either layer.

To ensure the data produced by a grain instance persists, it is backed up to the global data layer during the (backup) publication of the locally produced event message. This backup is facilitated with another flag value (between 'processing' and 'finished' described in §3.2) to indicate the start of the data transfer to the global data layer. This value contains the data's metadata (i.e., name, size, hash), which is checked during the recovery process to decide whether an event message needs to be processed: if the metadata in the flag matches the metadata of the actual data in the global data layer, the event message was successfully processed but the 'finished' flag could not be published by the failed host; otherwise, this event message needs to be processed again.

5 Implementation

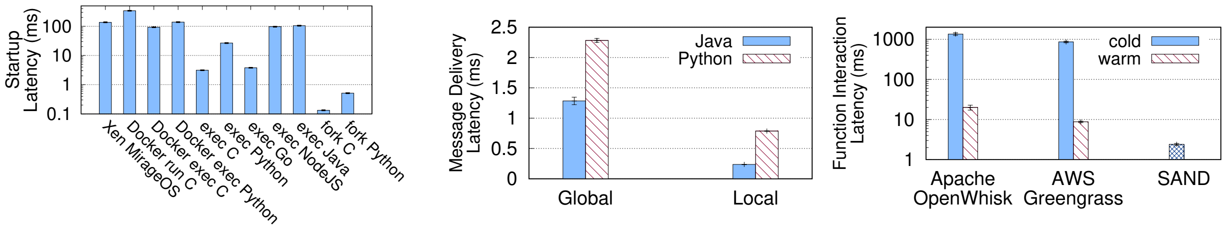
We implemented a complete SAND system with all components described in §4. Our system uses Docker [17] for application sandboxes, Apache Kafka [3] for the global message bus, Apache Cassandra [2] for the global data layer, and nginx [44] as the load balancer for the frontend server instances. We use these components off-the-shelf.

In addition, we implemented the host agent (7,273 lines of Java), the Python grain worker (732 lines of Python) and the frontend server (461 lines of Java). The host agent coordinates the local and global message buses and data layers, as well as manages application sandboxes and grains, by interacting with Kafka, Cassandra and Docker. The grain worker becomes dedicated to a specific grain after loading its code and necessary libraries, interacts with the local message bus, and forks grain instances for each associated event message. We use Apache Thrift [6] to automatically generate the interfaces for our Java implementations of the local message bus and the local data layer. The frontend server accepts connections handed over by the nginx load balancer, interacts with Kafka to deliver user requests into SAND and return application responses back to users. The frontend server embeds Jetty [32] as the HTTP endpoint and employs its thread pool to handle user requests efficiently.

For easy development and testing, we also implemented a SAND emulator (764 lines of Python) that supports SAND's API and logging for debugging. Developers can write their grains and workflows, and test them using the emulator before the actual deployment.

6 Evaluation

We evaluate SAND and compare it to Apache OpenWhisk [5] and AWS Greengrass [7]. We choose these two systems because we can run local installations for a fair comparison. We first report on microbenchmarks of alternative sandboxing mechanisms and SAND's hierarchical message bus. We then evaluate function interaction



(a) Function startup latencies.

(b) Message delivery latencies.

(c) Python function interaction latencies.

Figure 5: Measurements regarding function startup latencies, message delivery latencies and Python function interaction latencies, with error bars showing the 95% confidence interval.

latencies and the memory footprints of function executions, as well as investigate the trade-off between allocated memory and latency. Finally, we revisit the image processing pipeline, which was discussed as a motivational example in §1. We conducted all experiments on machines equipped with Intel Xeon E5520 with 16 cores at 2.27GHz and 48GB RAM, unless otherwise noted.

6.1 Sandboxing and Startup Latency

There are several alternative sandboxing mechanisms to isolate the applications and function executions (see §9). Here, we explore the startup latency of these alternatives.
Methodology. We measured the startup time until a function starts executing, with various sandboxing mechanisms including Docker (CE-17.11 with runc 1.0.0) and unikernel (Xen 4.8.1 with MirageOS 2.9.1), as well as spawning processes in C (gcc 4.8.5), Go (1.8.3), Python (2.7.13), NodeJS (6.12) and Java (1.8.0.151). We used an Intel Xeon E5-2609 host with 4 cores at 2.40GHz and 32GB RAM running CentOS 7.4 (kernel 4.9.63).
Results. Figure 5a shows the mean startup latencies with 95% confidence interval. Starting a process in a running, warm container via the Docker client interface (Docker exec C) is much faster than launching both the container and the process (Docker run C). Nonetheless, Docker adds significant overhead to function starts compared to starts without it (exec C, exec Python). Function starts with a unikernel (Xen MirageOS) are similar to using a container. Not surprisingly, spawning processes with binaries (exec C, exec Go) are faster than interpreted languages (exec Python, exec NodeJS) and Java, and forking processes (fork C, fork Python) is fastest among all.

6.2 Hierarchical Message Bus

Instead of a single unified message bus, SAND utilizes a local message bus on every host for fast function interactions. Here, we show the benefits of this approach.
Methodology. We created two processes on the same host that communicate in a producer-consumer style under load-free conditions. With Python and Java clients, we measured the latency for a message delivered via the global message bus (Kafka 0.10.1.0, 3 hosts, 3 replicas, default settings) and via our local message bus.
Results. Figure 5b shows the mean message delivery latencies with 95% confidence interval. The Python client (used by our grain instances) can deliver an event message to the next grain via the local message bus 2.90× faster than via the global message bus. Similarly, the Java client (used by our host agent) gets a 5.42× speedup.

6.3 Function Interaction Latency

Given two functions in a workflow, the function interaction latency is the time between the first function's finish and the second function's start.
Methodology. We created two Python functions, F_1 and F_2, such that F_1 produces an event message for F_2 to consume. We logged high-resolution timestamps at the end of F_1 and at the start of F_2. We used an Action Sequence in OpenWhisk [48], matching MQTT topic subscriptions in Greengrass [59] and SAND's workflow description. We then triggered F_1 with a request generator.

Recall that SAND uses a single, running container for multiple functions of an application. For a fair comparison, we measure function interaction latencies in OpenWhisk and Greengrass with warm containers. For completeness, we also report their cold call latencies, where a function call causes a new container to be launched.
Results. Figure 5c shows that SAND incurs a significantly shorter (Python) function interaction latency. SAND outperforms OpenWhisk and Greengrass, both with warm containers, by 8.32× and 3.64×, respectively. Furthermore, SAND's speedups are 562× and 358× compared to OpenWhisk and Greengrass with cold containers.

6.4 Memory Footprint

Concurrent calls to a function on today's serverless computing platforms are handled by concurrent execution instances. These instances are served either by launching

Table 1: Workloads and burst parameters.

Load	Rate (calls/min)	Duration (s)	Frequency (s)
A	1,000	8	240
B	250	8	240
C	1,000	30	120
D	1,000	8	120
E	250	30	120

new containers or by assigning them to warm containers if available. Because concurrently-running containers occupy system resources, we examine the memory footprint of such concurrent function executions.

Methodology. We made up to 50 concurrent calls to a single Python function. We ensured that all calls were served in parallel, and measured each platform's memory usage via `docker stats` and `ps` commands.

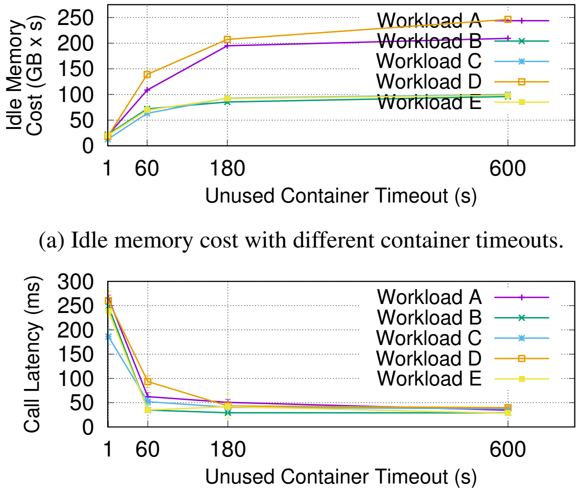
Results. We find that both OpenWhisk and Greengrass show a linear increase in memory footprint with the number of concurrent calls.[4] Each call adds to the memory footprint about 14.61MB and 13.96MB in OpenWhisk and Greengrass, respectively. In SAND, each call only adds 1.1MB on top of the 16.35MB consumed by the grain worker. This difference is because SAND forks a new process inside the same sandbox for each function call, whereas OpenWhisk and Greengrass use separate containers for concurrent calls.

6.5 Idle Memory Cost vs. Latency

Many serverless platforms use warm containers to prevent cold startup penalties for subsequent calls to a function. On the other hand, these platforms launch new containers when there are concurrent calls to a function but no warm containers available (§6.4). These new containers will also be kept warm until a timeout, occupying resources. Here, we investigate the trade-off between occupied memory and function call latency.

Methodology. We created 5 synthetic workloads each with 2,000 function calls. In all workloads, the call arrival time and the function processing time (i.e., busy wait) follow a Poisson distribution with a mean rate of 100 calls per minute and a mean duration of 600ms. To see how the serverless platforms behave under burst, we varied three parameters as shown in Table 1: 1) burst rate, 2) burst duration, and 3) burst frequency.

We explored 4 different unused-container timeouts in OpenWhisk. Unfortunately, this timeout cannot be modified in Greengrass, so we could not use it in this experiment. We computed the idle memory cost by multiplying the allocated but unused memory of the container instances with the duration of their allocations (i.e., total idle memory during the experiment). We reused the memory footprint results from §6.4. In OpenWhisk, the number of container instances depends on the concurrency, whereas SAND uses a single application sandbox.



(a) Idle memory cost with different container timeouts.

(b) Call latency with different container timeouts.

Figure 6: Idle memory cost vs. function call latency.

Results. Figures 6a and 6b show the effects of container timeouts in OpenWhisk on the idle memory cost and the function call latency, respectively. We observe that a long timeout is not suited for bursty traffic, with additional containers created to handle concurrent calls in a burst but are not needed afterwards. Even with a relatively short timeout of 180 seconds, the high idle memory costs suggest that containers occupy system resources without using them during the majority of our experiment. We also observe that a shorter timeout lowers the idle memory cost but leads to much longer function call latencies due to the cold start effect, affecting between 18.15%–33.35% of all calls in all workloads with a 1 second timeout. Interestingly, the frequent cold starts cause OpenWhisk to overestimate the number of required containers, partially offsetting the lowered idle memory cost achieved by shorter timeouts.

In contrast, SAND reduces idle memory cost from 3.32× up to two orders of magnitude with all workloads without sacrificing low latency (15.87–16.29 ms). SAND, by its sandboxing mechanism, handles concurrent calls to a function (or multiple functions) on a single host by forking parallel processes inside a container; therefore, SAND does not suffer from cold startup penalties. With higher load, SAND would amortize the penalty of starting a new sandbox on another host by using it both for multiple executions of a single function and for different functions. Our ongoing work includes intelligent monitoring and scheduling for additional sandboxes.

[4] In Greengrass, this relationship continues until 25 concurrent calls, after which calls get queued as shown in system logs.

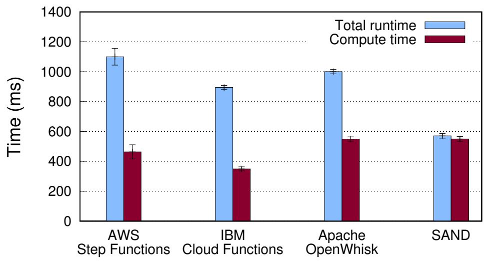


Figure 7: Total runtime and compute time of the image processing pipeline. Results show the mean values with 95% confidence interval over 10 runs, after discarding the initial (cold) execution.

6.6 Image Processing Pipeline

Here, we revisit our motivational example used in §1, i.e., the image processing pipeline. This pipeline consists of four consecutive Python functions, and is similar to the reference architecture used by AWS Lambda [51]. It first extracts the metadata of an image using ImageMagick [30]. A subset of this metadata is then verified and retained by the next function in the pipeline. The third function recognizes objects in the image using the SqueezeNet deep learning model [28] executed on top of the MXNet framework [4, 43]. Names of the recognized objects are appended to the extracted metadata and passed to the final function, which generates a thumbnail of the image and stores the metadata in a separate file.

Methodology. Each function recorded timestamps at the start and end of its execution, which we used to produce the actual compute time. The difference between the total time and the compute time gave each platform's overhead. The image was always read from a temporary local storage associated with each function call. We ran the pipeline on AWS Step Functions, IBM Cloud Functions, Apache OpenWhisk with Action Sequences, and `SAND`.

Results. Figure 7 shows the runtime breakdown of these platforms. Compared to other platforms, we find that `SAND` achieves the lowest overhead for running a series of functions. For example, `SAND` reduces the overhead by 22.0× compared to OpenWhisk, even after removing the time spent on extra functionality not supported in `SAND` yet (e.g., authentication and authorization). We notice that OpenWhisk re-launches the Python interpreter for each function invocation, so that libraries are loaded before a request can be handled. In contrast, `SAND`'s grain worker loads a function's libraries only once, which are then shared across forked grain instances handling the requests. The difference in compute times can be explained by the difference across infrastructures: `SAND` and OpenWhisk ran in our local infrastructure and produced similar values, whereas we had no control over AWS Step Functions and IBM Cloud Functions.

7 Experience with `SAND`

During and after the implementation of `SAND`, we also developed and deployed several applications on it. Here, we briefly describe these applications to show that `SAND` is general and can serve different types of applications.

The first application we developed is a simple web server serving static content (e.g., html, javascript, images) via two grains in a workflow. The first grain parses user requests and triggers another grain according to the requested file type. The second grain retrieves the file and returns it to our frontend server, which forwards it to the user. Our `SAND` web server has been in use since May 2017 to serve our lab's website. The second `SAND` application is the management service of our `SAND` system. The service has 19 grains, connects with the GUI we developed, and enables developers to create grains as well as to deploy and test workflows.

In addition, we made `SAND` available to researchers in our lab. They developed and deployed applications using the GUI and the management service. One group prototyped a simple virus scanner, whereby multiple grains executing in parallel check the presence of viruses in an email. Another group developed a stream analytics application for Twitter feeds, where grains in a workflow identify language, remove links and stop words, and compile a word frequency list to track the latest news trend.

8 Related Work

Despite its recency, serverless computing has already been used in various scenarios including Internet of Things and edge computing [7, 16], parallel data processing [33, 34], data management [14], system security enhancement [13], and low-latency video processing [20]. Villamizar et al. [56] showed that running applications in a serverless architecture is more cost efficient than microservices or monoliths. One can expect that serverless computing is going to attract more attention.

Beside commercial serverless platforms [1, 11, 15, 25, 29, 31], there have also been academic proposals for serverless computing. Hendrickson et al. [24] proposed OpenLambda after identifying problems in AWS Lambda [1], including long function startup latency and little locality consideration. McGrath et al. [42] also investigated latencies in existing serverless frameworks. These problems are important for serverless application development, where function interaction latencies are crucial. In `SAND`, we address these problems via our application sandboxing approach, as well as the hierarchical message queuing and storage mechanisms.

Other approaches also targeted the long startup latency problem. Slacker [23] identifies packages that are critical when launching a container. By prioritizing these packages and lazily loading others, it can reduce the container startup latency. This improvement would benefit serverless computing platforms that launch functions with cold starts. In SAND, an application sandbox is launched once per host for multiple functions of the same application, which amortizes a container's startup latency over time.

Pipsqueak [46] and its follow-up work SOCK [47] create a cache of pre-warmed Python interpreters, so that functions can be launched with an interpreter that has already loaded the necessary libraries. However, many functions may need the same (or a similar) interpreter, requiring mechanisms to pick the most appropriate one and to manage the cache. SAND does not use any sharing nor cache management schemes; a SAND grain worker is a dedicated process for a single function and its libraries.

McGrath et al. [41] proposed a queuing scheme with workers announcing their availability in warm and cold queues, where containers can be reused and new containers can be created, respectively. Unlike SAND, this scheme maps a single container per function execution.

9 Discussion & Limitations

Performance Isolation & Load Balancing. In this paper, we reduce function interaction latencies via our sandboxing mechanism as well as the hierarchical message queuing and storage. SAND executes multiple instances of an application's functions in parallel as separate processes in the same container. This sandboxing mechanism enables a cloud operator to run many functions (and applications) even on a single host, with low idle memory cost and high resource efficiency. However, it is possible that grains in a sandbox compete for the same resources and interfere with each other's performance. A single host may also not have the necessary resources for multiple sandboxes. In addition, SAND's locality-optimized policy with the hierarchical queuing and storage might lead to sub-optimal load balancing.

SAND currently relies on the operating system to ensure that the grains (and sandboxes) running in parallel will receive their fair share of resources. As such, CPU time (or other resource consumption) rather than the wall clock time could be used for billing purposes. Nevertheless, competing grains and sandboxes may increase the latency an application experiences.

Non-fork Runtime Support. SAND makes a trade-off to balance performance and isolation by using process forking for function executions. The downside is that SAND currently does not support language runtimes without native forking (e.g., Java and NodeJS).

Alternative Sandboxing Mechanisms. SAND isolates applications with containers. Virtual machines (VMs), HyperContainers [27], gVisor [21] and CNTR [53] are viable alternatives. VMs provide a stronger isolation than containers, but may increase the maintenance effort for each application's custom VM and have long launch times. Unikernels [38,39] can also be used to isolate applications with custom system software compiled with the desired functionality. However, dynamically adding/removing a function requires a recompilation, affecting the flexibility of function assignment to a host. In contrast, containers provide fast launch times, flexibility to dynamically assign functions, and low maintenance effort because the OS is shared among all application containers on a host. Recently open-sourced gVisor [22] provides a stronger fault isolation than vanilla containers.

For function executions, SAND uses separate processes. Unikernels [35, 38, 39] have also been proposed to isolate individual functions in serverless environments. A bare-bones unikernel-based VM (e.g., LightVM [40]) can launch faster than a container to execute a function; however, its image size depends on the libraries loaded by each function, and thus, may impact startup latency. Other alternatives include light-weight contexts (LWCs) [37] and threads. Particularly, LWCs may provide the best of both worlds by being lighter than processes, but achieving stronger isolation than threads by giving a separate view of resources to each LWC. We plan to extend SAND with these alternative approaches.

10 Conclusion & Future Work

This paper introduced SAND, a novel serverless computing platform. We presented the design and implementation of SAND, as well as our experience in building and deploying serverless applications on it. SAND employs a new sandboxing approach, whereby stronger mechanisms such as containers are used to isolate different applications and lighter OS concepts such as processes are used to isolate functions of the same application. This approach enables SAND to allocate and deallocate resources for function executions much faster and more resource-efficiently than existing serverless platforms. Combined with our hierarchical message bus, where each host runs a local message bus to enable fast triggering of functions running on the same host, SAND reduces function interaction latencies significantly.

For future work, we plan to address the limitations discussed in §9. In particular, we plan to intelligently distribute application functions and sandboxes across many hosts to better balance the system load without sacrificing application latency.

11 Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Patrick Stuedi, for their insightful comments.

References

[1] Amazon. AWS Lambda - Serverless Compute. https://aws.amazon.com/lambda/.

[2] Apache Cassandra. https://cassandra.apache.org/.

[3] Apache Kafka. https://kafka.apache.org/.

[4] MXNet: A Scalable Deep Learning Framework. http://mxnet.incubator.apache.org/.

[5] Apache OpenWhisk is a serverless, open source cloud platform. http://openwhisk.apache.org/.

[6] Apache Thrift. https://thrift.apache.org/.

[7] AWS Greengrass. https://aws.amazon.com/greengrass/.

[8] AWS Lambda for Java Quodlibet Medium. https://medium.com/@quodlibet_be/aws-lambda-for-java-5d5e954d3bdf.

[9] AWS Lambda Limits - AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/limits.html.

[10] What is AWS Step Functions? http://docs.aws.amazon.com/step-functions/latest/dg/welcome.html.

[11] Azure FunctionsServerless Architecture — Microsoft Azure. https://azure.microsoft.com/en-us/services/functions/.

[12] Best Practices for Azure Functions — Microsoft Docs. https://docs.microsoft.com/en-us/azure/azure-functions/functions-best-practices.

[13] Bila, N., Dettori, P., Kanso, A., Watanabe, Y., and Youssef, A. Leveraging the serverless architecture for securing linux containers. In *1st International Workshop on Serverless Computing* (2017), pp. 401–404.

[14] Chard, R., Chard, K., Alt, J., Parkinson, D. Y., Tuecke, S., and Foster, I. T. Ripple: Home automation for research data management. In *1st International Workshop on Serverless Computing* (2017), pp. 389–394.

[15] Cloud Functions - Serverless Environment to Build and Connect Cloud Services — Google Cloud Platform. https://cloud.google.com/functions/.

[16] de Lara, E., Gomes, C. S., Langridge, S., Mortazavi, S. H., and Roodi, M. Hierarchical serverless computing for the mobile edge. In *IEEE/ACM Symposium on Edge Computing* (2016).

[17] Docker - Build, Ship and Run Any App, Anywhere. https://www.docker.com/.

[18] Why Edge Computing Market Will Grow 30 Percent by 2022. http://www.eweek.com/networking/why-edge-computing-market-will-grow-30-percent-by-2022.

[19] Ellis, A. Functions as a Service (FaaS). https://blog.alexellis.io/functions-as-a-service/, 2017.

[20] Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G., and Winstein, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI* (2017).

[21] Google/gVisor: Container Runtime Sandbox. https://github.com/google/gvisor.

[22] Google open sources gVisor, a sandboxed container runtime. https://techcrunch.com/2018/05/02/google-open-sources-gvisor-a-sandboxed-container-runtime/.

[23] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016).

[24] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[25] hook.io. https://hook.io/.

[26] Hu, Y. C., Patel, M., Sabella, D., Sprecher, N., and Young, V. Mobile edge computinga key technology towards 5g. *ETSI white paper 11*, 11 (2015), 1–16.

[27] Hyper: Make VM run like Container. https://hypercontainer.io/.

[28] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360* (2016).

[29] Cloud Functions - Overview — IBM Cloud. https://www.ibm.com/cloud/functions.

[30] Convert, Edit, Or Compose Bitmap Images @ ImageMagick. https://www.imagemagick.org/.

[31] Iron.io - DevOps Solutions from Startups to Enterprise. https://www.iron.io/.

[32] Jetty - Servlet Engine and Http Server. http://www.eclipse.org/jetty/.

[33] Jonas, E. Microservices and Teraflops. http://ericjonas.com/pywren.html.

[34] Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., and Recht, B. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM.

[35] Koller, R., and Williams, D. Will serverless end the dominance of linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (2017), HotOS '17.

[36] Introducing Lambda@Edge in Preview Run Lambda functions at AWSs edge locations closest to your users. https://aws.amazon.com/about-aws/whats-new/2016/12/introducing-lambda-at-edge-in-preview-run-lambda-function-at-aws-edge-locations-closest-to-your-users/.

[37] Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., and Druschel, P. Light-weight contexts: an os abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).

[38] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. In *ASPLOS* (2013).

[39] Madhavapeddy, A., and Scott, D. J. Unikernels: the rise of the virtual library operating system. *Communications of the ACM* (2014).

[40] Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., and Huici, F. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17.

[41] McGrath, G., and Brenner, P. R. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)* (2017).

[42] McGrath, G., Short, J., Ennis, S., Judson, B., and Brenner, P. Cloud event programming paradigms: Applications and analysis. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)* (2016).

[43] GitHub - awslabs/mxnet-lambda: Reference Lambda function that predicts image labels for a image using an MXNet-built deep learning model. The repo also has pre-built MXNet, OpenCV libraries for use with AWS Lambda. `https://github.com/awslabs/mxnet-lambda`.

[44] nginx news. `https://nginx.org/`.

[45] Node.js. `https://nodejs.org/en/`.

[46] Oakes, E., Yang, L., Houck, K., Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Pipsqueak: Lean lambdas with large libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)* (2017).

[47] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[48] OpenWhisk Action Sequences - Create and invoke Actions. `https://console.bluemix.net/docs/openwhisk/openwhisk_actions.html#openwhisk_create_action_sequence`.

[49] PubNub: Making Realtime Innovation Simple. `https://www.pubnub.com/`.

[50] Satyanarayanan, M. The emergence of edge computing. *Computer 50*, 1 (2017), 30–39.

[51] GitHub - awslabs/lambda-refarch-imagerecognition: The Image Recognition and Processing Backend reference architecture demonstrates how to use AWS Step Functions to orchestrate a serverless processing workflow using AWS Lambda, Amazon S3, Amazon DynamoDB and Amazon Rekognition. `https://github.com/awslabs/lambda-refarch-imagerecognition/`.

[52] Shi, W., and Dustdar, S. The promise of edge computing. *Computer 49*, 5 (2016), 78–81.

[53] Thalheim, J., Bhatotia, P., Fonseca, P., and Kasikci, B. CNTR: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[54] UnitCluster - Make and run scripts in the cloud. `https://unitcluster.com/`.

[55] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), EuroSys '15.

[56] Villamizar, M., Garces, O., Ochoa, L., Castro, H. E., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., and Lang, M. Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In *CCGrid* (2016), pp. 179–182.

[57] Write-ahead logging. `https://en.wikipedia.org/wiki/Write-ahead_logging`.

[58] Webtask. `https://webtask.io/`.

[59] What is AWS Greengrass? `https://docs.aws.amazon.com/greengrass/latest/developerguide/what-is-gg.html`.

Cavs: An Efficient Runtime System for Dynamic Neural Networks

[1,2]Shizhen Xu†, [1,3]Hao Zhang†, [1,3]Graham Neubig, [3]Wei Dai, [1]Jin Kyu Kim, [2]Zhijie Deng,
[3]Qirong Ho, [2]Guangwen Yang, [3]Eric P. Xing

Carnegie Mellon University[1], Tsinghua University[2], Petuum Inc.[3] (†equal contributions)

Abstract

Recent deep learning (DL) models are moving more and more to dynamic neural network (NN) architectures, where the NN structure changes for every data sample. However, existing DL programming models are inefficient in handling dynamic network architectures because of: (1) substantial overhead caused by repeating dataflow graph construction and processing every example; (2) difficulties in batched execution of multiple samples; (3) inability to incorporate graph optimization techniques such as those used in static graphs. In this paper, we present "Cavs", a runtime system that overcomes these bottlenecks and achieves efficient training and inference of dynamic NNs. Cavs represents a dynamic NN as a static vertex function $\mathcal{F}$ and a dynamic instance-specific graph $\mathcal{G}$. It avoids the overhead of repeated graph construction by only declaring and constructing $\mathcal{F}$ once, and allows for the use of static graph optimization techniques on pre-defined operations in $\mathcal{F}$. Cavs performs training and inference by scheduling the execution of $\mathcal{F}$ following the dependencies in $\mathcal{G}$, hence naturally exposing batched execution opportunities over different samples. Experiments comparing Cavs to state-of-the-art frameworks for dynamic NNs (TensorFlow Fold, PyTorch and DyNet) demonstrate the efficacy of our approach: Cavs achieves a near one order of magnitude speedup on training of dynamic NN architectures, and ablations verify the effectiveness of our proposed design and optimizations.

1 Introduction

Deep learning (DL), which refers to a class of neural networks (NNs) with deep architectures, is now a workhorse powering state-of-the-art results on a wide spectrum of tasks [53, 54, 30]. One reason for its widespread adoption is the variety and quality of software toolkits, such as Caffe [23], TensorFlow [1], PyTorch [36] and DyNet [33, 34], which ease programming of DL models, and speed computation by harnessing modern computing hardware (e.g. GPUs), software libraries (e.g. CUDA, cuDNN [6]), and compute clusters [56, 57, 7].

One dominant programming paradigm, adopted by DL toolkits such as Caffe and TensorFlow, is to represent a neural network as a static dataflow graph [32, 1], where computation functions in the NN are associated with nodes in the graph, and input and output of the computation map to edges. It requires DL programmers to define the network architecture (i.e. the dataflow graph) using symbolic expressions, once before beginning execution. Then, for a given graph and data samples, the software toolkits can automatically derive the correct algorithm for training or inference, following backpropagation [21] and auto-differentiation rules. With proper optimization, the execution of these static dataflow graphs can be highly efficient; as the dataflow graph is fixed for all data, the evaluation of multiple samples through one graph can be naturally batched, leveraging the improved parallelization capability of modern hardware (e.g. GPUs). Moreover, by separating model declaration and execution, it makes it possible for the graph to be optimized once at declaration time [1], with these optimizations benefiting the efficiency of processing arbitrary input data batches at execution time.

While the dataflow graph has major efficiency advantages, its applicability highly relies on a key assumption – the graph (i.e. NN architecture) is fixed throughout the runtime. This assumption however breaks for *dynamic* NNs, where the network architectures conditionally change with every input sample, such as NNs that compute over sequences of variable lengths [22, 43], trees [45], and graphs [26].

Due to the growing interest in these sorts of dynamic models, recent years have seen an increase in the popularity of frameworks based on *dynamic declaration* [49, 33, 11], which declare a different dataflow graph per sample. While dynamic declaration is convenient to developers as it removes the restriction that computation be completely specified before training begins, it exhibits a few limitations. First, constructing a graph for every sample results in substantial overhead, which grows linearly with the number of input instances. In fact, we find graph construction takes longer time than the computation in some frameworks (see §5.2). It also prevents the application of complex static graph optimization techniques (see §3.4). Moreover, since each sample owns a dataflow graph specifying its unique computational pattern, batching together similarly shaped computations across instances is non-trivial. Without batching, the computation is inefficient due to its lack of ability to exploit modern computational hardware. While some progress has been made in recent research [34, 27], how to automatically batch the computational operations from different graphs remains a difficult problem.

To address these challenges, we present Cavs, an efficient runtime system for dynamic NNs that exploits the recurrent and recursive nature of dynamic NNs. Instead of declaring a dataflow graph per sample, it decomposes a dynamic NN into two components: a static vertex function $\mathcal{F}$ that is only declared (by the user) and optimized once before execution, and an input-specific graph $\mathcal{G}$ obtained via I/O at runtime. Cavs inherits the flexibility of symbolic programming [1, 12, 33] for DL; it requires users to define $\mathcal{F}$ by writing symbolic expressions in the same way as in static declaration. With $\mathcal{F}$ and $\mathcal{G}$, the workflow of training or testing a dynamic NN is cast as scheduling the execution of $\mathcal{F}$ following the structure of the input graph $\mathcal{G}$. Cavs will perform auto-differentiation, schedule the execution following dependencies in $\mathcal{G}$, and guarantee efficiency and correctness.

Cavs' design allows for highly efficient computation in dynamic graphs for a number of reasons. First, it allows the vertex function only to be defined and constructed once for any type of structured data, hence avoiding the overhead of repeated dataflow graph construction. Second, as the dataflow graph encoded by the vertex function is static throughout the runtime, it can benefit from various static graph optimizations [1, 5, 12, 18](§3.4), which is not the case in the scenario of dynamic declaration (§2.2). Moreover, it naturally exposes opportunities for batched computation, i.e. we are able to parallelize the execution of $\mathcal{F}$ over multiple vertices from different input graphs (§3.2) with the support of our proposed memory management strategy (§3.3).

To evaluate Cavs' performance, we compare it to several state-of-the-art systems supporting dynamic NNs. We focus our experiments on GPU training, and verify that both Fold and DyNet suffer from substantial overhead caused by repeated graph preprocessing or construction, which is bypassed by Cavs (§5.2). In a comparison with unbatched dynamic graphs in PyTorch and DyNet, two widely-used dynamic NN libraries, we verify that batching is essential for efficient processing. In a comparison with TensorFlow Fold and DyNet Autobatching, two libraries that allow for the use of dynamic NNs with automatic operation batching, we find that Cavs' has significant performance advantages; on static graphs it performs equivalently or slightly better, and on dynamic NNs with difficult-to-batch workloads (e.g. Tree-LSTM [45] and Tree-FC [27]), Cavs demonstrates near one order of magnitude speedups across multiple dataset and hyper-parameter settings (§5.1). We further investigate the effectiveness of our design choices: Cavs benefits from not only our proposed memory management strategy, but also various optimizations on graph execution, which were originally for static dataflow graphs and not applicable in dynamic declaration.

To summarize, we make three primary contributions in this paper: (1) We propose a novel representation for dynamic NNs, based on which we design four APIs and implement the Cavs runtime system (§3.1); (2) We propose several novel strategies in Cavs for efficient training and inference of dynamic NNs: the batching policy (§3.2), a memory management mechanism to guarantee the memory coalescing (§3.3), and multiple graph execution optimizations (§3.4); (3) We compare Cavs to state-of-the-art systems for dynamic NNs (§5). We reveal the problems of existing systems, and report near 10x speedup for Cavs on various experimental settings. We also verify the effectiveness of our proposed design strategies, and quantize their contributions to the final performance.

2 Background

2.1 Dynamic Neural Networks

Successful NN models generally exhibit suitable architectures that capture the structures of the input data. For example, convolutional neural networks [24, 53], which apply fixed-structured operations to fixed-sized images, are highly effective precisely because they capture the spatial invariance common in computer vision domains [39, 44]. However, apart from images, many forms of data are structurally complex and can not be readily captured by fixed-structured NNs. Appropriately reflecting these structures in the NN design has shown effective in sentiment analysis [45], semantic similarity between sentence pairs [40], and image segmentation [26].

To see this, we will take the constituency parsing problem as an example. Sentences in natural languages are often represented by their constituency parse tree [45, 31], whose structure varies depending on the content of the sentence itself (Fig. 1(a)). Constituency parsing is an important problem in natural language processing that aims to determine the corresponding grammar type of all internal nodes given the parsing tree of a sentence. Fig. 1(b) shows an example of a network that takes into account this syntactic structure, generating representations for the sentence by traversing the parse tree bottom-up and combining the representations for each sub-tree

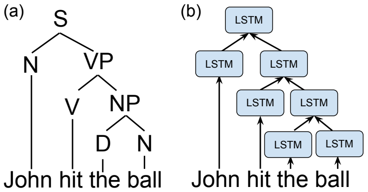


Figure 1: An example of a dynamic NN: (a) a constituency parsing tree, (b) the corresponding Tree-LSTM network. We use the following abbreviations in (a): S for sentence, N for noun, VP for verb phrase, NP for noun phrase, D for determiner, and V for verb.

using a dynamic NN called Tree Structured Long Short-term Memory (Tree-LSTM) [45]. In particular, each node of the tree maps to a LSTM function [22]. The internal computations and parameters of the LSTM function is defined in Fig. 4. At each node, it takes a variable number of inputs and returns to the parent node a vector representing the parsing semantics up to that point, until the root LSTM node returns a vector representing the semantics of the entire sentence.

The important observation is that the NN structure varies with the underlying parsing tree over *each* input sample, but the same LSTM cell is constant in shape and repeated at each internal node. Similar examples can be found for graph input [25, 26] and sequences of variable lengths [43, 2]. We refer to these NNs that exhibit different structures for different input samples as *dynamic neural networks*, in contrast to the static networks that have fixed network architecture for all samples.

2.2 Programming Dynamic NNs

There is a natural connection between NNs and directed graphs: we can map the graph nodes to the computational operations or parameters in NNs, and let the edges indicate the direction of the data being passed between the nodes. In this case, we can represent the process of training NNs as batches of data flowing through computational graphs, i.e. *dataflow graphs* [3, 1, 33].

Static declaration. As mentioned previously, *static declaration* is one dominant programming paradigm for programming NNs [3, 1, 5]. Fig 2(a) summarizes its workflow, which assumes all data samples share a fixed NN structure declared symbolically in a dataflow graph $\mathcal{D}$. Static declaration, using a single dataflow graph $\mathcal{D}$, cannot express dynamic NNs with structures changing with data samples. A primary remedy to this problem is to forgo the efficiency gains of static dataflow graphs and instead use a *dynamic declaration* framework.

Dynamic declaration. Fig 2(b) illustrates the workflow of dynamic declaration. By creating a unique dataflow graph $\mathcal{D}_k^p$ for each sample x_k^p according to its associated structure, dynamic declaration is able to express sample-dependent dataflow graphs. It however causes extra overhead on graph construction and puts constraints on runtime optimization, which usually lead to inefficient execution. Particularly, since a dataflow graph $\mathcal{D}_k^p$ needs to be constructed per sample, the overhead is linearly increasing with the number of samples, and sometimes yields downgraded performance [27] (§5.2), even for frameworks with optimized graph construction implementations [33]. Moreover, we can hardly benefit from any well-established dataflow graph optimization (§3.4). We will have to perform graph processing/optimization for each dataflow graph and every single sample; but incorporating this optimization itself has a non-negligible overhead. More importantly, as we are unable to batch the computation of different structured graphs, we note in Fig 2(b) single-instance computation $\mathcal{D}_k^p(x_k^p)$ would be very inefficient in the absence of batched computation.

Dynamic batching. To address the batching problem, some recent effort, notably TensorFlow Fold [27] and DyNet [34], propose *dynamic batching* that dynamically groups similarly shaped operations from different graphs, and batch their execution whenever possible.

Fold turns dynamic dataflow graphs into a static control flow graph to enable batched execution, but introduces a complicated functional programming-like interface and a large graph preprocessing overhead. As we will show in §5.2, the graph construction sometimes slows down the computation by 4x. DyNet proposes an auto-batching strategy that searches for batching opportunities by profiling every fine-grained operator, while this step itself has non-negligible overhead (§5.2). It is also not open to dataflow graph level optimizations.

In summary, there are three major challenges that prevent the efficient execution of dynamic neural networks: (1) non-negligible graph construction overhead; (2) difficulties in parallel execution; (3) unavailability to graph execution optimization.

2.3 Motivation

Our motivation for Cavs comes from a key property of dynamic NNs: most dynamic NNs are designed to exhibit a recursive structure; Within the recursive structure, a static computational function is being applied following the topological order over instance-specific graphs. For instance, if we denote the constituency parsing tree in §2.1 as a graph $\mathcal{G}$, where each node of the tree maps to a vertex in $\mathcal{G}$, we note the Tree-LSTM can be interpreted as follows: a computational cell function, specified in advance, is applied from leaves to the root, following the dependencies in $\mathcal{G}$. $\mathcal{G}$ might change with input samples, but the cell function itself is always static: It is parametrized by a fixed set of learnable parameters and interacts in the same way with its neighbors when applied at different vertices of $\mathcal{G}$.

These observations motivate us to decompose a dynamic NN into two parts: (1) a static computational *vertex function* $\mathcal{F}$ that needs to be declared by the program-

```
/* (a) static declaration */
// all samples must share one graph
declare a static dataflow graph $\mathcal{D}$.
for $p = 1 \to P$:
  read the $p$th data batch $\{x_k^p\}_{k=1}^K$.
  batched computation: $\mathcal{D}(\{x_k^p\}_{k=1}^K)$.
```

```
/* (b) dynamic declaration */
for $p = 1 \to P$:
  read the $p$th data batch $\{x_k^p\}_{k=1}^K$.
  for $k = 1 \to K$:
    declare a dataflow graph $\mathcal{D}_k^p$ for $x_k^p$.
    single-instance computation: $\mathcal{D}_k^p(x_k^p)$.
```

```
/* (c) our proposed vertex-centric model */
declare a symbolic vertex function $\mathcal{F}$.
for $p = 1 \to P$:
  read the $p$th data batch $\{x_k^p\}_{k=1}^K$.
  read their associated graphs $\{\mathcal{G}_k^p\}_{k=1}^K$.
  compute $\mathcal{F}$ over $\{\mathcal{G}_k^p\}_{k=1}^K$ with inputs $\{x_k^p\}_{k=1}^K$.
```

Figure 2: The workflows of (a) static declaration, (b) dynamic declaration, (c) Cavs. Notations: $\mathcal{D}$ notates both the dataflow graph itself and the computational function implied by it; p is the index of a batch while k is the index of a sample in the batch.

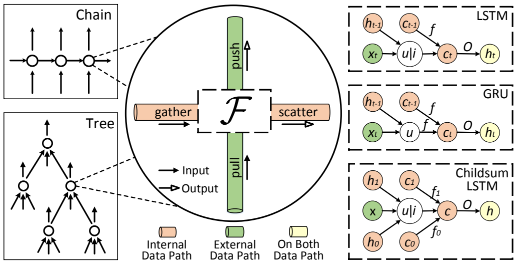


Figure 3: Cavs represents a dynamic structure as a dynamic input graph $\mathcal{G}$ (left) and a static vertex function $\mathcal{F}$ (right).

mer once before runtime; (2) a dynamic *input graph* $\mathcal{G}$ that changes with every input sample[1]. With this representation, the workflow of training a dynamic NN can be cast as scheduling the evaluation of the symbolic construct encoded by $\mathcal{F}$, following the graph dependencies of $\mathcal{G}$, as illustrated in Fig 2(c). This representation exploits the property of dynamic NNs to address the aforementioned issues in the following ways:

Minimize graph construction overhead. Cavs only requires users to declare $\mathcal{F}$ using symbolic expressions, and construct it once before execution. This bypasses repeated construction of multiple dataflow graphs, avoiding overhead. While it is still necessary to create an I/O function to read input graphs $\mathcal{G}$ for each sample, this must be done by any method, and only once before training commences, and it can be shared across samples.

Batched execution. With the proposed representation, Cavs transforms the problem of evaluating data samples $\{x_k^p\}_{k=1}^K$ (at the pth batch) on different dataflow graphs $\{\mathcal{D}_k^p\}_{k=1}$ [27, 34] into a simpler form – scheduling the execution of the vertex function $\mathcal{F}$ following the dependencies in input graphs $\{G_k^p\}_{k=1}$. For the latter problem, we can easily batch the execution of $\mathcal{F}$ on multiple vertices at runtime (§3.2), leveraging the batched computational capability of modern hardware and libraries.

Open to graph optimizations. Since the vertex function $\mathcal{F}$ encodes a dataflow graph which is static throughout runtime, it can benefit from various graph optimizations originally developed for static declaration, such as kernel fusion, streaming, and our proposed lazy batching, which are not effective in dynamic declaration.

Based on this motivation, we next describe the Cavs system. Cavs faces the following challenges in system design: (1) how to design minimal APIs in addition to the symbolic programming interface to minimize user code; (2) how to schedule the execution of $\mathcal{F}$ over multiple input graphs to enable batched computation; (3) how to manage memory to support the dynamic batching; (4) how to incorporate static graph optimization in Cavs's execution engine to exploit more parallelism.

3 Cavs Design and Optimization

3.1 Programming Interface

Conventional dataflow graph-based programming models usually entangle the computational workflow in $\mathcal{F}$ with the structure in $\mathcal{G}$, and require users to express them as a whole in a single dataflow graph. Instead, Cavs separates the static vertex function $\mathcal{F}$ from the input graph $\mathcal{G}$ (see Fig 3). While users use the same set of symbolic operators [1, 11] to assemble the computational workflow in $\mathcal{F}$, Cavs proposes four additional APIs, `gather`, `scatter`, `pull`, `push`, to specify how the messages shall be passed between connected vertices in $\mathcal{G}$:

- `gather(child_idx)`: `gather` accepts an index of a child vertex, gets its output, and returns a list of symbols that represent the output of the child.
- `scatter(op)`: `scatter` reverses `gather`. It sets the output of the current vertex as `op`. If this vertex is `gather`ed, the content of `op` will be returned.

`gather` and `scatter` are motivated by the GAS model in graph computing [14] – both are vertex-centric APIs that help users express the overall computational patterns by thinking locally like a vertex: `gather` receives messages from dependent vertices, while `scatter` updates information to parent vertices (see discussion in §6).

However, in dynamic NNs, the vertex function $\mathcal{F}$ usually takes input from not only the internal vertices of $\mathcal{G}$ (internal data path in Fig 3), but also the external environment, e.g. an RNN can take inputs from a CNN feature extractor or some external I/O (external data path in Fig 3). Cavs therefore provides another two APIs to express such semantics:

- `pull()`: `pull` grabs inputs from the external of the current dynamic structure, e.g. another NN, or I/O.
- `push(op)`: `push` reverses `pull`. It sets the output of the current vertex as `op`. If this vertex is `pull`ed by others, the content of `op` will be returned.

[1] In the following text, we will distinguish the term *vertex* from *node*. We use *vertex* to denote a vertex in the input graph while *node* to denote an operator/variable in a dataflow graph. Hence, a vertex function can have many nodes as itself represents a dataflow graph.

```
def ℱ():
  for k in range(N):
    S = gather(k) # gather states of child vertices
    c_k, h_k = split(S, 2) # get hidden states c and h
  x = pull() # pull the first external input x

  # specify the computation
  h = ∑_{k=0}^{N−1} h_k
  i = sigmoid(W^(i)× x + U^(i)× h + b^(i))
  for k in range(N):
    f_k = sigmoid(W^(f)× x + U^(f)× h_k + b^(f))
  o = sigmoid(W^(o)× x + U^(o)× h + b^(o))
  u = tanh(W^(u)× x + U^(u)× h + b^(u))
  c = i ⊗ u + ∑_{k=0}^{N−1} f_k ⊗ c_k
  h = o ⊗ tanh(c)

  scatter(concat([c, h], 1)) # scatter c, h to parents
  push(h)                    # push to external connectors
```

Figure 4: The vertex function of an N-ary child-sum TreeLSTM [45] in Cavs. Within $\mathcal{F}$, users declare a computational dataflow graph using symbolic operators. The defined $\mathcal{F}$ will be evaluated on each vertex of $\mathcal{G}$ following graph dependencies.

Once $\mathcal{F}$ declared, together with an input graph $\mathcal{G}$, they encode a recursive dataflow graph structure, which maps to a subgraph of the implicit full dataflow graph of the model that may needs to be explicitly declared in traditional programming models. Via push and pull, Cavs allows users to connect any external static dataflow graph to a dynamic structure encoded by $(\mathcal{F}, \mathcal{G})$, to express more complex model architectures, such as the LRCN [9] (i.e. connecting a CNN to an RNN), or an encoder-decoder LSTM network [43] (i.e. connecting two different recursive structures). With these four APIs, we present in Fig 4 an example user program how the N-ary child-sum Tree-LSTM [45] can be simply expressed by using them and other mathematical operators.

Auto-differentiation. Given a vertex function $\mathcal{F}$ Cavs derives $\partial\mathcal{F}$ following the auto-differentiation rules: for each math expression such as $s_l = \mathrm{op}(s_r)$ in $\mathcal{F}$, Cavs generates a corresponded backward expression $\nabla s_r = \texttt{grad\_op}(\nabla s_l, s_l, s_r)$ in $\partial\mathcal{F}$. For the four proposed operators, we note scatter is the gradient operator of gather in the sense that if gather collects inputs from child vertex written by scatter at the forward pass, a scatter needs to be performed to write the gradients for its dependent vertices to gather at the backward pass. Hence, for an expression like $s_l = \texttt{gather}(\texttt{child\_idx})$ in $\mathcal{F}$, Cavs will generate a backward expression $\texttt{scatter}(\nabla s_l)$ in $\partial\mathcal{F}$. Similarly, the gradient operator of scatter is gather. The same rules apply for push and pull.

Expressiveness. With these four APIs, Cavs can be seen as a middle ground between static and dynamic declaration. In the best case that the NN is fully recursive (e.g. most recurrent or recursive NNs), it can be represented by a single vertex function and an input graph. While in the worst case, that every sample has a unique input graph while every vertex in the graph has a unique way to interact with its neighboring vertices (i.e. the NN is dynamic but non-recursive), Cavs reduces to dynamic declaration that one has to define a vertex function for each vertex of each input graph. Fortunately, dynamic NNs in this scenario are usually avoided because of the difficulties in design, programming and learning.

3.2 Scheduling

Once $\mathcal{F}$ is defined and $\mathcal{G}$ is obtained from I/O, Cavs will perform computation by scheduling the evaluation of $\mathcal{F}$ over data samples $\{x_i\}_{i=1}^N$ and their input graphs $\{\mathcal{G}_i\}_{i=1}^N$.

Forward pass. For a sample x_i with its input graph $\mathcal{G}_i$, the scheduler starts the forward pass from the input vertices of $\mathcal{G}_i$, and proceeds following the direction indicated by the edges in $\mathcal{G}_i$: at each sub-step, the scheduler figures out the next activated vertex in $\mathcal{G}_i$, and evaluates all expressions in $\mathcal{F}$ at this vertex. It then marks this vertex as *evaluated*, and proceeds with the next activated vertex until reaching a terminal vertex (e.g. the loss function). A vertex of $\mathcal{G}$ is activated if and only if all its dependent vertices have been evaluated.

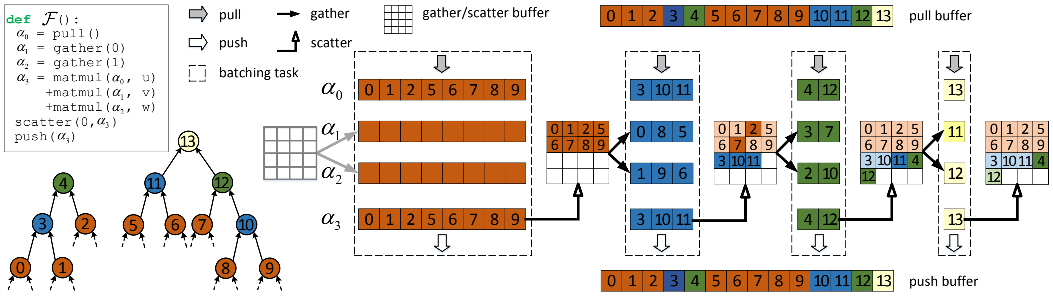
Backward pass. The backward pass is continued right after the forward. The scheduler first resets the status of all vertices as *not evaluated*, then scans the graph in a reverse direction, starting from the ending point of the forward pass. It evaluates $\partial\mathcal{F}$ at each vertex until all vertices have been evaluated in the backward pass.

To train a NN to convergence, the above process has to be iterated on all samples $\{x_i\}_{i=1}^N$ and their input graphs $\{\mathcal{G}_i\}_{i=1}^N$, for many epochs. We next describe our batched execution policy to speed the computation.

Batching policy. Given a data batch $\{x_k\}_{k=1}^K \subseteq \{x_i\}_{i=1}^N$ and associated graphs $\{\mathcal{G}_k\}_{k=1}^K$, this policy groups multiple vertices and performs batched evaluation of $\mathcal{F}$ in order to reduce kernel launches and exploit parallelism. Specifically, a forward pass over a batch $\{x_k\}_{k=1}^K$ are performed in multiple steps. At each step t, Cavs analyzes $\{\mathcal{G}_k\}_{k=1}^K$ at runtime and determines a set V_t that contains all activated vertices in graphs $\{\mathcal{G}_k\}_{k=1}^K$. It then evaluates $\mathcal{F}$ over these vertices by creating a *batched execution task*, with the task ID set to t[2]. The task is executed by the Cavs execution engine (§3.4). Meanwhile, the scheduler records this task by pushing V_t into a stack $\mathcal{S}$. To perform backward pass, the scheduler pops out an element V_t from $\mathcal{S}$ at each step – the execution engine will evaluate the derivative function $\partial\mathcal{F}$ over vertices in V_t, until all vertices of $\{\mathcal{G}_k\}_{k=1}^K$ are evaluated.

We note the batching policy is similar to the *dynamic batching* in Fold [27] and DyNet [33]. However, Cavs determines how to batch fully dynamically during runtime using simple breadth-first search with negligible cost (instead of analyzing full dataflow graphs before every iteration of the execution). Since batched computation requires the inputs to an expression over multiple

[2] Whenever the context is clear, we use V_t to denote both the set of vertices to be batched together, and the batched execution task itself.



```
def 𝓕():
  α₀ = pull()
  α₁ = gather(0)
  α₂ = gather(1)
  α₃ = matmul(α₀, u)
      +matmul(α₁, v)
      +matmul(α₂, w)
  scatter(0,α₃)
  push(α₃)
```

Figure 5: The memory management at the forward pass of $\mathcal{F}$ (top-left) over two input trees (bottom-left). Cavs first analyzes $\mathcal{F}$ and inputs – it creates four dynamic tensors $\{\alpha_n\}_{n=0}^{3}$, and figures out there will be four batch tasks (dash-lined boxes). Starting from the first task (orange vertices $\{0,1,2,5,6,7,8,9\}$), Cavs performs batched evaluation of each expression in $\mathcal{F}$. For example, for the pull expression $\alpha_0 = \texttt{pull}()$, it indexes the content of α_0 on all vertices from the *pull buffer* using their IDs, and copies them to α_0 continuously; for scatter and push expressions, it scatters a copy of the output (α_3) to the *gather buffer*, and pushes them to the push buffer, respectively. Cavs then proceeds to the next batching task (blue vertices). At this task, Cavs evaluates each expression of $\mathcal{F}$ once again for vertices $\{3,10,11\}$. (e.g. for a pull expression $\alpha_0 = \texttt{pull}()$, it pulls the content of α_0 from pull buffer again; for a gather expression $\alpha_2 = \texttt{gather}(1)$ at vertex 3, it gathers the output of the second child of 3, which is 1); it writes results continuously at the end of each dynamic tensor. It proceeds until all batching tasks are finished.

vertices to be placed on a continuous memory buffer, we develop a new memory management support for it.

3.3 Memory Management

In static declaration [1, 33], a symbol in the user program usually corresponds to a fixed-sized *tensor* object with a batch size dimension. While in Cavs, each batching task V_t is determined at runtime. For the batched computation to be efficient, Cavs must guarantee for each batching task, the inputs to each expression of $\mathcal{F}$ over a group of runtime-determined vertices coalescing in memory.

```
struct DynamicTensor {
  vector<int> shape;
  int bs;
  int offset;
  void* p; };
```

Figure 6: Dynamic tensor.

Cavs proposes a novel data structure *dynamic tensor* to address this challenge (Fig 6). A dynamic tensor is a wrapper of a multi-dimensional array [1, 52]. It contains four attributes: `shape`, `bs`, a pointer `p` to a chunk of memory, and `offset`. `shape` is an array of integers representing the specific shape of the tensor excluding the batch dimension. It can be inferred from the user program and set before execution. The batch size `bs` is dynamically set by the scheduler at runtime at the beginning of a batching task. To access a dynamic tensor, one moves `p` forward with the value of `offset`, and reads/writes number of elements equal to $\texttt{bs} \cdot \prod_i \texttt{shape}[i]$. Therefore, `bs` together with `offset` provide a view of the tensor, and the state of the tensor will vary based on their values. Given a vertex function $\mathcal{F}$, Cavs creates dynamic tensors $\{\alpha_n\}_{n=1}^{N}$ for each non-parameter symbol $s_n (n = 1, \ldots, N)$ in $\mathcal{F}$, and also $\{\nabla\alpha_n\}_{n=1}^{N}$ as their gradients, while it creates static tensors for model parameters.

Fig 5 illustrates how the memory is assigned during the forward pass by manipulating dynamic tensors. In particular, in a training iteration, for a batching task V_t, the scheduler sets `bs` of all $\{\alpha_n\}_{n=1}^{N}$ to $M_t = |V_t|$ (the number of vertices in V_t). The execution engine then performs batched evaluation of each expression in $\mathcal{F}$. For an expression $s_l = \texttt{op}(s_r)$[3], Cavs first accesses α_r (the dynamic tensor of the RHS symbol s_r) – it offsets $\alpha_r.\texttt{p}$ by $\alpha_r.\texttt{offset}$, and reads a block of $M_t \prod_i \alpha_r.\texttt{shape}[i]$ elements, and presents it as a tensor with batch size M_t and other dimensions as $\alpha_r.\texttt{shape}$. It then applies batched computational kernels of the operator `op` over this memory block, and writes the results to α_l (the dynamic tensor of the LHS symbol s_l) on the continuous block in between $[\alpha_l.\texttt{p} + \alpha_l.\texttt{offset}, \alpha_l.\texttt{p} + \alpha_l.\texttt{offset} + M_t \prod_i \alpha_l.\texttt{shape}[i]]$. Upon the completion of V_t, the scheduler increases `offset` of all $\{\alpha_n\}_{n=1}^{N}$ by $M_t \prod_i \alpha_n.\texttt{shape}[i]$, respectively. It then starts the next task V_{t+1}. Hence, intermediate results generated in each batching task at forward pass are stored continuously in the dynamic tensors, and their offsets are recorded.

At the entrance of $\mathcal{F}$, the vertices $\{v_m\}_{m=1}^{M_t}$ in V_t need to interact with its dependent vertices in previous V_{t-1} to `gather` their outputs as inputs (L3 of Figure 4), or `pull` inputs from the external (L5 of Figure 4). Cavs maintains memory buffers to enable this (Figure 5). It records the offsets of the dynamic tensors for each $v_m \in V_t$, and therefore during the execution of `gather` operator, the memory slices of specific children can be indexed. As shown in Figure 5, `gather` and `scatter` share the same temporary buffer for memory re-organization, but `push` and `pull` operate on external memory buffers.

Algorithm 1 summarizes the memory management during forward pass. The backward execution follows an exactly reverse order of the forward pass (§3.2), which we skip in the text. With this strategy, Cavs guarantees memory continuity for any batched computation of $\mathcal{F}$ and $\partial\mathcal{F}$. Compared to dynamic batching in DyNet, Cavs performs memory movement only at the entrance

[3] Note that the user-defined expressions can be arbitrary, e.g. with more than one argument or return values

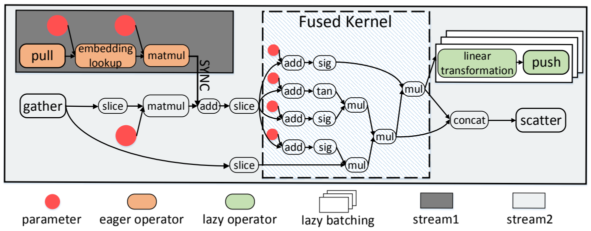


Figure 7: The dataflow graph encoded by $\mathcal{F}$ of Tree-LSTM.

and exit of $\mathcal{F}$, instead of for each expression (operator). We empirically find this significantly reduces overhead of memory operations (§5.3).

Algorithm 1 Memory management at forward pass.

```
function FORWARD($\{V_t\}_{t=1}^{T}, \{\alpha_n\}_{n=1}^{N}, \mathcal{F}$)
  for $t = 1 \to T$ do
    for $n = 1 \to N$ do $\alpha_n$.bs $\leftarrow M_t$ end for
    for each expression like $s_l = \texttt{op}(s_r)$ in $\mathcal{F}$ do
      if op $\in \{\texttt{gather}, \texttt{pull}\}$ then
        $C \leftarrow \prod_i \alpha_l.\texttt{shape}[i], q \leftarrow \alpha_l.\texttt{p} + \alpha_l.\texttt{offset}$.
        for $v_m \in V_t (m = 1 \to M_t)$ do
          $\texttt{src} \leftarrow \texttt{IndexBuffer}(\texttt{op}, m), \texttt{dest} \leftarrow q + (m-1)C$.
          $\texttt{memcpy}(\texttt{dest}, \texttt{src}, C)$.
        end for
      else if op $\in \{\texttt{scatter}, \texttt{push}\}$ then
        $C \leftarrow \prod_i \alpha_r.\texttt{shape}[i], q \leftarrow \alpha_r.\texttt{p} + \alpha_r.\texttt{offset}$.
        for $v_m \in V_t (m = 1 \to M_t)$ do
          $\texttt{dest} \leftarrow \texttt{IndexBuffer}(\texttt{op}, m), \texttt{src} \leftarrow q + (m-1)C$.
          $\texttt{memcpy}(\texttt{dest}, \texttt{src}, C)$.
        end for
      else
        perform batched computation: $\alpha_l = \texttt{op\_kernel}(\alpha_r)$.
      end if
    end for
    for $n = 1 \to N$ do $\alpha_n.\texttt{offset} += M_t \prod_i \alpha_n.\texttt{shape}[i]$ end for
  end for
end function
```

3.4 Optimizing Execution Engine

Since Cavs separates out a static dataflow graph encoded by $\mathcal{F}$, we can replace the original $\mathcal{F}$ with an optimized one that runs more efficiently, as long as maintaining correctness. We next described our optimization strategies.

Lazy batching and streaming[4]**.** In addition to batched execution of $\mathcal{F}$, the lazy batching and streaming explore potential parallelism for a certain group of finer-grained operators in $\mathcal{F}$ or $\partial\mathcal{F}$ called lazy and eager operators.

Definition. An operator in $\mathcal{F}$ ($\partial\mathcal{F}$) is a *lazy operator* if at the forward (backward) pass, for $\forall v \in \mathcal{G}, \forall \mathcal{G} \in \{\mathcal{G}_k\}_{k=1}^{K}$, the evaluation of $\mathcal{F}$ ($\partial\mathcal{F}$) at any parent (dependent) vertex of v does not rely on the evaluation of $\mathcal{F}$ at v. It is an *eager operator* if the evaluation at v does not rely on the evaluation of $\mathcal{F}$ ($\partial\mathcal{F}$) at any dependents (parents) of v.

Proposition. Denote $\mathcal{D}_{\mathcal{F}}$ ($\mathcal{D}_{\partial\mathcal{F}}$) as the dataflow graph encoded by $\mathcal{F}$ ($\partial\mathcal{F}$), and $g, s \in \mathcal{D}_{\mathcal{F}}$ ($\mathcal{D}_{\partial\mathcal{F}}$) as nodes of `gather` and `scatter` operator, respectively. A node that has g as its dependent and is not on any path from g to s is a lazy operator. A node that has s as its ancestor and is not on any path from g to s is an eager operator.

Fig 7 illustrates a forward dataflow graph of the vertex function of Tree-LSTM, with eager and lazy operators colored. A property of them is that their evaluation is not fully subject to the dependency reflected by the input graph $\mathcal{G}$. For instance, the `pull` operator in Fig 7 is eager and can be executed in prior – even before $\mathcal{F}$ has been evaluated at the vertices that `gather` tries to interact with; the `push` operator is lazy, so we can defer its execution without impacting the evaluation of $\mathcal{F}$ at parent vertices. Similarly, in $\partial\mathcal{F}$, the gradient derivation for model parameters are mostly lazy – their execution can be deferred as long as the gradients of hidden states are derived and propagated in time. Cavs leverages this property and proposes the lazy batching strategy. It defers the execution of all lazy operators in $\mathcal{F}$ and $\partial\mathcal{F}$ until all batching tasks $\{V_t\}_{t=1}^{T}$ has finished. It then performs a batched execution of these lazy operators over all vertices of $\{\mathcal{G}_k\}_{k=1}^{K}$. These operators includes, but is not limited to, the `push` operator that is doing memory copy, and operators for computing gradients of model parameters. Lazy batching helps exploit more parallelism and significantly reduces kernel launches. Empirically lazy batching brings 20% overall improvement (§5.3).

To leverage the exhibited parallelization opportunity between eager operators and the operators on the path from `gather` to `scatter` (Figure 7), Cavs proposes a streaming strategy that pipelines the execution of these two groups of operators. It allocates two streams, and puts the eager operators on one stream, and the rest (excluding lazy operators) on the other. Hence, independent operators in two streams run in parallel, while for those operators that depend on an eager operator, this dependency is respected by synchronization barriers (Fig 7).

Automatic kernel fusion. Given $\mathcal{F}$, before execution, Cavs will run a *fusion detector* [20] to scan its corresponded dataflow graph and report all *fuse-able* subgraphs therein, i.e. all nodes in a fuse-able subgraph can be fused as a single operator that behaves equivalently but takes less execution time (e.g. with fewer kernel launches and I/O, or faster computation). Currently, we only detect groups of directly linked elementwise operators, such as $+$, `sigmoid`, as shown in Fig 7, and we use a simple union-find algorithm to detect the largest possible fuse-able subgraphs. Given a fuse-able subgraph, Cavs adopts de facto automatic code generation techniques [37, 8, 38, 35] to generate lower-level kernel implementations. Replacing the original fuse-able subgraphs with fused operators during execution is beneficial in many aspects: (1) it reduces the number of kernel launches; (2) on some devices such as GPUs, kernel fu-

[4] Streaming is a borrowed terminology from CUDA programming which means executing different commands concurrently with respect to each other on different GPU streams. As Cavs' optimizations are agnostic to the low-level hardware, we use streaming interchangeably with multi-threading if the underlying computing hardware is CPU.

sion transform device memory access into faster device registers access. We empirically report another 20% improvement with automatic kernel fusion (§5.3).

4 Implementation

Cavs is implemented as a C++ library and integrable with existing DL frameworks to enhance their support for dynamic NNs. It is composed of three major layers (which is the case for most popular frameworks [3, 1, 33]): (1) a frontend that provides device-agnostic symbolic programming interface; (2) an intermediate layer that implements the core execution logic; (3) a backend with device-specific kernels for all symbolic operators.

Frontend. In addition to the four APIs, Cavs provides a macro operator `VertexFunction`. Users instantiate it by writing symbolic expressions and specifying methods to read input graphs. It encapsulates scatter/gather semantics, so users can continue using higher level APIs. To construct more complex NN architectures (e.g. encoder-decoder LSTM [43], LRCN [9])), users employ `push` and `pull` to connect multiple vertex functions, or to external structures.

Intermediate Layer. Cavs has its core runtime logic at this layer, i.e. the batching scheduler, the memory management, and the execution engine, etc.

Backend. Following practice [1, 33, 12], we implement device-specific operator kernels at this layer. Cavs has optimized implementations for the four proposed operators (`gather`, `scatter`, `pull`, `push`). Specifically, `gather` and `pull` index different slices of a tensor and puts them together continuously on memory; `scatter` and `push` by contrast splits a tensor along its batch dimension, and copy different slices to different places. Cavs implements customized memcpy kernels for there four operators, so that copying multiple slices from (or to) different places is performed within one kernel.

Distributed Execution. While Cavs's implementations are focused on improving the efficiency on a single node, they are compatible with most data-parallel distributed systems for deep learning [56, 7, 1], and can also benefit distributed execution on multiple nodes.

5 Evaluation

In this section, we evaluate Cavs on multiple NNs and datasets, obtaining the following major findings: (1) Cavs has little overhead: on static NNs, Cavs demonstrates equal performance on training and inference with other systems; On several NNs with notably difficult-to-batch structures, Cavs outperforms all existing frameworks by a large margin. (2) We confirm the graph construction overhead is substantial in both Fold [27] and dynamic declaration [33], while Cavs bypasses it by loading input graphs through I/O. (3) We verify the effectiveness of our proposed design and optimization via ablation studies, and discuss Cavs' advantages over other DL systems for dynamic dataflow graphs.

Environment. We perform all experiments in this paper on a single machine with an NVIDIA Titan X (GM200) GPU, a 16-core CPU, and CUDA v8.0 and cuDNN v6 installed. As modern DL models are mostly trained using GPUs, we focus our evaluation on GPUs, but note Cavs' design and implementation do not rely on a specific type of device. We mainly compare Cavs to TensorFlow v1.2 [1] with XLA [18] and its variant Fold [27], PyTorch v0.3.0 [11], and DyNet v2.0 [33] with autobatching [34], as they have reported better performance than other frameworks [5, 50] on dynamic NNs. We focus on metrics for system performance, e.g. time to scan one epoch of data. Cavs produces exactly the same numerical results with other frameworks, hence the same per-epoch convergence

Models and dataset. We experiment on the following models with increasing difficulty to batch: (a) `Fixed-LSTM` language model (LM): a static sequence LSTM with fixed steps for language modeling [42, 43, 55]. We train it using the PTB dataset [48] that contains over 10K different words. We set the number of steps as 64, i.e. at each iteration of training, the model takes a 64-word sentence from the training corpus, and predicts the next word of each word therein. Obviously, the computation can be by nature batched easily, as each sentence has exactly the same size. (b) `Var-LSTM` LM: that accepts variable-length inputs. At each iteration the model takes a batch of natural sentences with different length from PTB, and predicts the next words; (c) `Tree-FC`: the benchmarking model used in [27] with a single fully-connected layer as its cell function. Following the same setting in [27], we train it over synthetic samples generated by their code [47] – each sample is associated with a complete binary tree with 256 leaves (therefore 511 vertices per graph); (d) `Tree-LSTM`: a family of dynamic NNs widely adopted for text analysis [26, 51]. We implement the binary child-sum Tree-LSTM in [45], and train it as a sentiment classifier using Stanford sentiment treebank (SST) dataset [40]. The dataset contains 8544 training sentences, each associated with a human annotated grammar tree, and the longest one has 54 words.

5.1 Overall Performance

We first verify the viability of our design on the easiest-to-batch case: `Fixed-LSTM` language model. We compare Cavs to the following three strong baselines: (1) `CuDNN` [6]: a CuDNN-based fixed-step sequence LSTM, which is highly optimized by NVIDIA using handcrafted kernels and stands as the best performed implementation on NVIDIA GPUs; (2) `TF`: the official implementation of `Fixed-LSTM` LM in TensorFlow repository [46] based on static declaration; (3) `DyNet`: we implement a 64-step

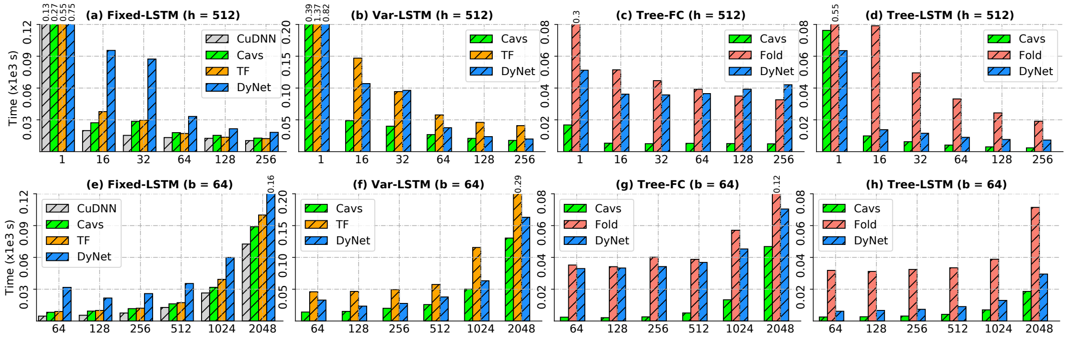


Figure 8: Comparing five systems on the averaged time to finish one epoch of training on four models: `Fixed-LSTM`, `Var-LSTM`, `Tree-FC` and `Tree-LSTM`. In (a)-(d) we fix the hidden size *h* and vary the batch size *bs*, while in (e)-(h) we fix *bs* and vary *h*.

LSTM in DyNet based on dynamic declaration – we declare a dataflow graph per sample, and train with the autobatching [34] enabled; (4) `Cavs` with batching policy, and all input samples have a same input graph – a 64-node chain. We train the model to converge, and report the average time per epoch in Fig 8(a)(e), where in (a) we fix the hidden size *h* of the LSTM unit as 512 and vary the batch size *bs*, and in (e) we fix $bs = 64$ and vary *h*. Empirically, CuDNN performs best in all cases, but note it is highly inflexible. Cavs performs slightly better than `TF` in various settings, verifying that our system has little overhead handling fully static graphs, though it is specialized for dynamic ones. We also conclude from Fig 8 that batching is essential for GPU-based DL: $bs = 128$ is nearly one order of magnitude faster than $bs = 1$ regardless of used frameworks. For Cavs, the batching policy is 1.7x, 3.8x, 7.0x, 12x, 15x, 25x, 36x faster than non-batched at $bs = 2, 4, 8, 16, 32, 64, 128$, respectively.

Next, we experiment with `Var-LSTM`, the most commonly used RNN for variable-length sequences. We compare the following three implementations (CuDNN-based LSTM cannot handle variable-length inputs): (1) `TF`: an official TensorFlow implementation based on the dynamic unroll approach described in §6; (2) `DyNet`: an official implementation from DyNet benchmark repository based on dynamic declaration [10]; (3) `Cavs`: where each input sentence is associated with a chain graph that has number of vertices equal to the number of words. We vary *h* and *bs*, and report the results in Figure 8(b)(f), respectively. Although all three systems perform batched computation in different ways, `Cavs` is consistently 2-3 times faster than `TF`, and outperforms `DyNet` by a large margin. Compared to `TF`, `Cavs` saves computational resources. `TF` dynamically unrolls the LSTM unit according to the longest sentence in the current batch, but it cannot prevent unnecessary computation for those sentences that are shorter than the longest one.

We then turn to `Tree-FC`, a dynamic model for benchmarking. Since vanilla TensorFlow is unable to batch its computation, we compare `Cavs` to (1) `DyNet` and (2) `Fold`, a specialized library built upon TensorFlow for dynamic NNs, with a depth-based dynamic batching strategy. To enable the batching, it however needs to preprocess the input graphs, translate them into intermediate representations and pass them to lower-level TensorFlow control flow engine for execution. We report the results in Figure 8(c)(g) with varying *bs* and *h*, respectively. For all systems, we allocate a single CPU thread for graph preprocessing or construction. `Cavs` shows at least an order of magnitude speedups than `Fold` and `DyNet` at $h \leq 512$. Because the size of the synthetic trees is large, one major advantage of `Cavs` over them is the alleviation of graph preprocessing/construction overhead. With a single CPU thread, `Fold` takes even more time on graph preprocessing than computation (§5.3).

Finally, we compare three frameworks on `Tree-LSTM` in Figure 8(d)(h): `Cavs` is 8-10x faster than `Fold`, and consistently outperforms `DyNet`. One difference in this experiment is that we allocate as many CPU threads as possible (32 on our machine) to accelerate graph preprocessing for `Fold`, otherwise it will take much longer time. Further, we note `DyNet` performs much better here than on `Tree-FC`, as the size of the input graphs in SST (maximally 54 leaves) is much smaller than the synthetic ones (256 leaves each) in `Tree-FC` experiments. We observe `DyNet` needs more time on graph construction for large input graphs, and `DyNet`'s dynamic batching is less effective on larger input graphs, as it has to perform frequent memory checks to support its dynamic batching, which we will discuss in §5.3. We also compare Cavs with PyTorch – its per-epoch time on Tree-LSTM is 542s, 290x slower than Cavs when the batch size is 256. Compared to other systems, PyTorch cannot batch the execution of dynamic NNs.

5.2 Graph Construction and Computation

In this section, we investigate the graph construction overhead in Fold and DyNet. To batch computation of different graphs, Fold analyzes the input graphs to recognize batch-able dynamic operations, then translates them into intermediate instructions, with which, TensorFlow generates appropriate control flow graphs for evaluation – we will treat the overhead caused in both steps as

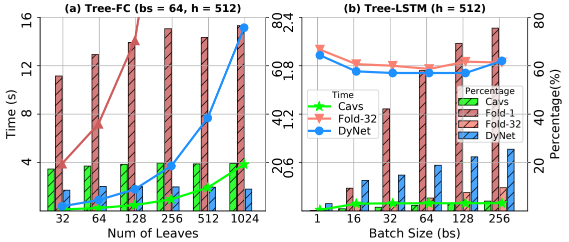


Figure 9: The averaged graph construction overhead per epoch when training (a) Tree-FC with different size of input graphs (b) Tree-LSTM with different batch size. The curves show absolute time in second (left y-axis), and the bar graphs show its percentage of the overall time (right y-axis).

Fold's graph construction overhead. DyNet, as a typical dynamic declaration framework, has to construct as many dataflow graphs as the number of samples. Though DyNet has optimized its graph construction to make it lightweight, the overhead still grows with the training set and the size of input graphs. By contrast, Cavs takes constant time to construct a small dataflow graph encoded by $\mathcal{F}$, then reads input graphs through I/O. To quantify the overhead, we separate the graph construction from computation, and visualize in Figure 9(a) how it changes with the average number of leaves (graph size) of input graphs on training Tree-FC, with fixed $bs = 64, h = 512$. We compare (1) Cavs (2) Fold-1 which is Fold with one graph processing thread and (3) DyNet. We plot for one epoch, both the (averaged) absolute time for graph construction and it percentage of the overall time. Clearly we find that all three systems take increasingly more time when the size of the input graphs grows, but Cavs, which loads graphs through I/O, causes the least overhead at all settings. In terms of the relative time, Fold unfortunately wastes 50% at 32 leaves, and 80% when the tree has 1024 leaves, while DyNet and Cavs take only 10% and 20%, respectively.

We also wonder how the overhead is related with batch size when there is fixed computational workload. We report in Figure 9(b) the same metrics when training Tree-LSTM with varying bs. We add another baseline Fold-32 with 32 threads for Fold's graph preprocessing. As Fold-1 takes much longer time than others, we report its time at $bs = 1, 16, 32, 64, 128, 256$ here (instead of showing in Figure 9): 1.1, 7.14, 31.35, 40.1, 46.13, 48.77. Except $bs = 1$, all three systems (except Fold-1) take almost constant time for graph construction in one epoch, regardless of bs, while Fold-32 and DyNet take similar time, but Cavs takes 20x less. Nevertheless, at the percentage scale, increasing bs makes this overhead more prominent, because larger batch size yields improved computational efficiency, therefore less time to finish one epoch. This, from one perspective, reflects that the graph construction is a main obstacle that grows with the number of training samples and prevents the efficient training of dynamic NNs in existing frameworks, while Cavs successfully overcomes this barrier.

# leaves	time (s)	Speedup	bs	time (s)	Speedup
32	**0.6** / 3.1 / 4.1	**5.4** / 7.1	1	76 / 550 / **62**	**7.2** / 0.8
64	**1.1** / 3.9 / 8.0	3.7 / 7.5	16	**9.8** / 69 / 12	7.0 / 1.2
128	**2** / 6.2 / 16	3.0 / 7.9	32	**6.2** / 43 / 9.9	7.0 / 1.6
256	**4** / 10.6 / 33.7	2.7 / 8.7	64	**4.1** / 29 / 7.4	**7.2** / 1.8
512	**8** / 18.5 / 70.6	2.3 / 8.9	128	**2.9** / 20.5 / 5.9	7.1 / 2.0
1024	**16** / 32 / 153	2.1 / **9.7**	256	**2.3** / 15.8 / 5.4	7.0 / **2.4**

Table 1: The averaged computation time (Cavs/Fold/DyNet) and the speedup (Cavs vs Fold/DyNet) for training one epoch on Tree-FC with varying size of the input trees (left part), and on Tree-LSTM with varying batch size (right part).

Apart from the graph construction we report in Table 1 the computation-only time. Cavs shows maximally 5.4x/9.7x and 7.2x/2.4x speedups over Fold/DyNet on Tree-FC and Tree-LSTM, respectively. The advantages stem from two main sources: an optimized graph execution engine, and a better-suited memory management strategy, which we investigate next.

5.3 Optimizations

Graph Execution Engine. To reveal how much each optimization in §3.4 contributes to the final performance, we disable lazy batching, fusion and streaming in Cavs and set this configuration as a baseline (speedup = 1). We then turn on one optimization at a time and record how much speedup it brings. We train Fixed-LSTM and Tree-LSTM, and report the averaged speedups one computation-only time in one epoch over the baseline configuration in Fig 10, with $bs = 64$ but varying h. Lazy batching and fusion consistently deliver nontrivial improvement – lazy batching is more beneficial with a larger h while fusion is more effective at smaller h, which are expected: lazy batching mainly parallelizes matrix-wise operations (e.g. matmul) commonly with $O(h^2)$ or higher complexity, while fusion mostly works on elementwise operations with $O(h)$ complexity [19].

Streaming, compared to the other strategies, is less effective on Tree-LSTM than on Fixed-LSTM, as we have found the depth of the input trees in SST exhibit high variance, i.e. some trees are much deeper than others. In this case, many batching tasks only have one vertex to be evaluated. The computation is highly fragmented and the efficiency is bounded by kernel launching latency. Lazy batching and fusion still help as they both reduce kernel launches (§3.4). Streaming, which tries to pipeline multiple kernels, can hardly yield obvious improvement.

Memory Management. Cavs' performance advantage also credits to its memory management that reduces memory movements while guarantees continuity. Quantitatively, it is difficult to compare Cavs to Fold, as Fold relies on TensorFlow where memory management is highly coupled with other system aspects. Qualitatively, we find Cavs requires less memory movement (e.g. memcpy) during dynamic batching. Built upon the tf_while operator, whenever Fold performs depth-

Programming Model	**Frameworks**	**Expressiveness**	**Batching**	**Graph Construction Overhead**	**Graph Optimization**
static declaration	Caffe, TensorFlow	×	√	low	beneficial
dynamic declaration (eager evaluation)	PyTorch, Chainer	√	×	N/A	unavailable
dynamic declaration (lazy evaluation)	DyNet	√	√	high	limited benefits
Fold	TensorFlow-Fold	√	√	high	unknown
Vertex-centric	Cavs	√	√	low	beneficial

Table 2: A side-by-side comparison of existing programming models for dynamic NNs, and their advantages and disadvantages.

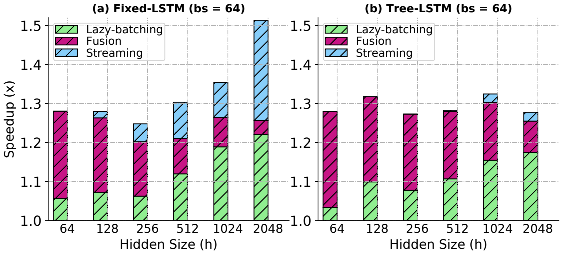


Figure 10: Improvement of each optimization strategy on execution engine over a baseline configuration (speedup = 1).

bs	Memory operations (s) (*Cavs / DyNet*)		Computation (s) (*Cavs / DyNet*)	
	Train	Inference	Train	Inference
16	**1.14** / 1.33	**0.6** / 1.33	**9.8** / 12	**2.9** / 8.53
32	**0.67** / 0.87	**0.35** / 0.87	**6.1** / 9.8	**1.9** / 5.35
64	**0.39** / 0.6	**0.21** / 0.6	**4.0** / 7.4	**1.3** / 3.48
128	**0.25** / 0.44	**0.13** / 0.44	**2.9** / 5.9	**0.97** / 2.52
256	**0.17** / 0.44	**0.09** / 0.44	**2.3** / 5.4	**0.77** / 2.58

Table 3: Breakdowns of average time per epoch on memory-related operations and computation, comparing Cavs to DyNet on training and inference of `Tree-LSTM` with varying *bs*.

based batching at depth d, it has to move all the contents of nodes in the dataflow graphs at depth $d-1$ to a desired location, as the control flow does not support cross-depth memory indexing. This results in redundant memcpy, especially when the graphs are highly skewed. By contrast, Cavs only copies contents that are necessary to the batching task. DyNet has a specialized memory management strategy for dynamic NNs. Compared to Cavs, it however suffers substantial overhead caused by repeated checks of the memory continuity – whenever DyNet wants to batch operators with same signatures, it checks whether their inputs are continuous on memory [34]. The checking overhead increases with *bs* and is more prominent on GPUs. Thanks to the simplicity of both systems, we are able to profile the memory-related overhead during both training and inference, and separate it from computation. We compare them on `TreeLSTM`, and report the breakdown time per epoch in Table 3 under different *bs*. We observe the improvement is significant (2x - 3x) at larger *bs*, especially during inference where DyNet has its continuity checks concentrated.

6 Related Work

DL programming models. In addition to §2.2, we summarize in Table 2 the major programming models and frameworks for dynamic NNs, and their pros and cons, in contrast to Cavs. Within static frameworks, there are also efforts on adapting static declaration to support sequence RNNs, such as *static unrolling* [17], *bucketing* [15] and *dynamic unrolling* [16]. The ideas are to pad zero at the end of samples so that they have the same structure (i.e. same length) for batched computation. However, they all result in unnecessary computation and can not express more complex structures than sequences. Asynchronous model-parallelism [13] enables the concurrent execution of different graphs similar to batched execution in Cavs, it however may suffer from insufficient cache re-usage and overhead by multiple kernel launches (on GPUs).

Execution optimization. A variety of developed techniques from other areas (e.g. kernel fusion, constant folding) have been adapted to speed the computation of DL dataflow graphs [1, 5, 12, 18]. Cavs separates the static vertex function from the dynamic-varying input graph, so it benefits from most of the aforementioned optimizations. We learn from these strategies and reflect them in Cavs' execution engine. We further propose lazy batching and concurrent execution to exploit more parallelism exposed by our APIs.

Graph-based systems. The vertex-centric programming model has been extensively developed in graph computing [29, 14, 4, 41]. Cavs draws insights from the GAS model [14], but is fundamentally different: `gather` and `scatter` in Cavs are fully symbolic – they allow backpropagation through them; graph computing systems compute on large natural graphs, while Cavs addresses problems that each sample has a unique graph and the training is iterative on batches of samples. In terms of system design, Cavs also faces different challenges, such as scheduling for batched execution of different graphs, guaranteeing the memory continuity. There are also some graph-based ML systems, such as GraphLab [28], but they do not handle instance-based graphs, and do not offer batching advantages for dynamic DL workloads.

7 Conclusion

We present Cavs, an efficient system for dynamic neural networks. With a novel representation, designed scheduling policy, memory management strategy, and graph execution optimizations, Cavs avoids substantial graph construction overhead, allows for batched computation over different structured graphs, and can benefit from well-established graph optimization techniques. We compare Cavs to state-of-the-art systems for dynamic NNs and report a near one order of magnitude speedup across various dynamic NN architectures and settings.

References

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning. *arXiv preprint arXiv:1605.08695* (2016).

[2] BAHDANAU, D., CHO, K., AND BENGIO, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[3] BERGSTRA, J., BASTIEN, F., BREULEUX, O., LAMBLIN, P., PASCANU, R., DELALLEAU, O., DESJARDINS, G., WARDE-FARLEY, D., GOODFELLOW, I. J., BERGERON, A., AND BENGIO, Y. Theano: Deep Learning on GPUs with Python. In *NIPSW* (2011).

[4] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 1.

[5] CHEN, T., LI, M., LI, Y., LIN, M., WANG, N., WANG, M., XIAO, T., XU, B., ZHANG, C., AND ZHANG, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[6] CHETLUR, S., WOOLLEY, C., VANDERMERSCH, P., COHEN, J., TRAN, J., CATANZARO, B., AND SHELHAMER, E. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[7] CUI, H., ZHANG, H., GANGER, G. R., GIBBONS, P. B., AND XING, E. P. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 4.

[8] DAVE, C., BAE, H., MIN, S.-J., LEE, S., EIGENMANN, R., AND MIDKIFF, S. Cetus: A source-to-source compiler infrastructure for multicores. *Computer 42*, 12 (2009).

[9] DONAHUE, J., ANNE HENDRICKS, L., GUADARRAMA, S., ROHRBACH, M., VENUGOPALAN, S., SAENKO, K., AND DARRELL, T. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 2625–2634.

[10] DYNET VARIABLE LENGTH LSTM. `https://github.com/neulab/dynet-benchmark`.

[11] FACEBOOK. `http://pytorch.org/`.

[12] FACEBOOK OPEN SOURCE. Caffe2 is a lightweight, modular, and scalable deep learning framework. `https://github.com/caffe2/caffe2`, 2017.

[13] GAUNT, A., JOHNSON, M., RIECHERT, M., TARLOW, D., TOMIOKA, R., VYTINIOTIS, D., AND WEBSTER, S. Ampnet: Asynchronous model-parallel training for dynamic neural networks. *arXiv preprint arXiv:1705.09786* (2017).

[14] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs.

[15] GOOGLE. Tensorflow bucketing. `https://www.tensorflow.org/versions/r0.12/api_docs/python/contrib.training/bucketing`.

[16] GOOGLE. Tensorflow dynamic rnn. `https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn`.

[17] GOOGLE. Tensorflow static rnn. `https://www.tensorflow.org/api_docs/python/tf/nn/static_rnn`.

[18] GOOGLE TENSORFLOW XLA. `https://www.tensorflow.org/performance/xla/`.

[19] GUSTAFSON, J. L. Reevaluating amdahl's law. *Communications of the ACM 31*, 5 (1988), 532–533.

[20] GYSI, T., OSUNA, C., FUHRER, O., BIANCO, M., AND SCHULTHESS, T. C. Stella: A domain-specific tool for structured grid methods in weather and climate models. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for* (2015), IEEE, pp. 1–12.

[21] HINTON, G., DENG, L., YU, D., DAHL, G. E., MOHAMED, A.-R., JAITLY, N., SENIOR, A., VANHOUCKE, V., NGUYEN, P., SAINATH, T. N., ET AL. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine 29*, 6 (2012), 82–97.

[22] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[23] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093* (2014).

[24] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS* (2012).

[25] Liang, X., Hu, Z., Zhang, H., Gan, C., and Xing, E. P. Recurrent topic-transition gan for visual paragraph generation. *arXiv preprint arXiv:1703.07022* (2017).

[26] Liang, X., Shen, X., Feng, J., Lin, L., and Yan, S. Semantic object parsing with graph lstm. In *European Conference on Computer Vision* (2016), Springer, pp. 125–143.

[27] Looks, M., Herreshoff, M., Hutchins, D., and Norvig, P. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181* (2017).

[28] Low, Y., Gonzalez, J. E., Kyrola, A., Bickson, D., Guestrin, C. E., and Hellerstein, J. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041* (2014).

[29] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 135–146.

[30] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[31] Mitchell, D. C. Sentence parsing. *Handbook of psycholinguistics* (1994), 375–409.

[32] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 439–455.

[33] Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980* (2017).

[34] Neubig, G., Goldberg, Y., and Dyer, C. On-the-fly operation batching in dynamic computation graphs. *arXiv preprint arXiv:1705.07860* (2017).

[35] NVIDIA. `http://docs.nvidia.com/cuda/nvrtc/index.html`.

[36] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch.

[37] Quinlan, D. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters 10*, 02n03, 215–226.

[38] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices 48*, 6 (2013), 519–530.

[39] Simonyan, K., and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR* (2015).

[40] Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., and Potts, C. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing* (2013), pp. 1631–1642.

[41] Sundaram, N., Satish, N., Patwary, M. M. A., Dulloor, S. R., Anderson, M. J., Vadlamudi, S. G., Das, D., and Dubey, P. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1214–1225.

[42] Sundermeyer, M., Schlüter, R., and Ney, H. Lstm neural networks for language modeling. In *Thirteenth Annual Conference of the International Speech Communication Association* (2012).

[43] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (2014), pp. 3104–3112.

[44] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A., et al. Going deeper with convolutions.

[45] Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from

tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[46] TENSORFLOW FIXED-SIZED LSTM LANGUAGE MODEL. `https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/ptb_word_lm.py`.

[47] TENSORFLOW FOLD BENCHMARK CODE. `https://github.com/tensorflow/fold/tree/master/tensorflow_fold/loom/benchmarks`.

[48] THE PENN TREE BANK (PTB) DATASET. `http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz`.

[49] TOKUI, S., OONO, K., HIDO, S., AND CLAYTON, J. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)* (2015), vol. 5.

[50] TOKUI, S., OONO, K., HIDO, S., AND CLAYTON, J. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)* (2015).

[51] VINYALS, O., KAISER, Ł., KOO, T., PETROV, S., SUTSKEVER, I., AND HINTON, G. Grammar as a foreign language. In *Advances in Neural Information Processing Systems* (2015), pp. 2773–2781.

[52] WALT, S. V. D., COLBERT, S. C., AND VAROQUAUX, G. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering 13*, 2 (2011), 22–30.

[53] YAN, Z., ZHANG, H., JAGADEESH, V., DECOSTE, D., DI, W., AND PIRAMUTHU, R. Hd-cnn: Hierarchical deep convolutional neural network for image classification. *ICCV* (2015).

[54] YAN, Z., ZHANG, H., WANG, B., PARIS, S., AND YU, Y. Automatic photo adjustment using deep neural networks. *ACM Transactions on Graphics (TOG) 35*, 2 (2016), 11.

[55] ZAREMBA, W., SUTSKEVER, I., AND VINYALS, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[56] ZHANG, H., HU, Z., WEI, J., XIE, P., KIM, G., HO, Q., AND XING, E. Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines. *arXiv preprint arXiv:1512.06216* (2015).

[57] ZHANG, H., ZHENG, Z., XU, S., DAI, W., HO, Q., LIANG, X., HU, Z., WEI, J., XIE, P., AND XING, E. P. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 181–193.

DeepCPU: Serving RNN-based Deep Learning Models 10x Faster

Minjia Zhang* Samyam Rajbhandari* Wenhan Wang Yuxiong He
Microsoft Business AI and Research
{*minjiaz,samyamr,wenhanw,yuxhe*}@*microsoft.com*

Abstract

Recurrent neural networks (RNNs) are an important class of deep learning (DL) models. Existing DL frameworks have unsatisfying performance for online serving: many RNN models suffer from long serving latency and high cost, preventing their deployment in production.

This work characterizes RNN performance and identifies low data reuse as a root cause. We develop novel techniques and an efficient search strategy to squeeze more data reuse out of this intrinsically challenging workload. We build DeepCPU, a fast serving library on CPUs, to integrate these optimizations for efficient RNN computation. Our evaluation on various RNN models shows that DeepCPU improves latency and efficiency by an order of magnitude on CPUs compared with existing DL frameworks such as TensorFlow. It also empowers CPUs to beat GPUs on RNN serving. In production services of Microsoft, DeepCPU transforms many models from non-shippable (due to latency SLA violation) to shippable (well-fitting latency requirements) and saves millions of dollars of infrastructure costs.

1. Introduction

Deep learning (DL) is a fast-growing field pervasively influencing many applications on image, speech, and text processing. Traditional feed forward neural networks assume that all inputs (and outputs) are independent of each other. This could be a bad idea for many tasks. For example, to predict the next word in a sentence, we had better know which words come before that. To classify what kind of event is happening to the next point of a movie, we had better reason from the previous events. Recurrent neural networks (RNNs) are an important and popular class of DL models that address this issue by making use of sequential information [22,35,51]. RNNs perform the same task for every element in the sequence, with the output being dependent on the previous computation. This is somewhat similar to the human learning, e.g., to understand a document, we read word by word, sentence by sentence, and carry the information along in our memory while reading. RNNs have shown great promise in many natural language processing tasks, e.g., language model [16,44], machine translation [15,21,58], machine reading comprehension [18,25,39,53], speech recognition [31,34,66], and conversational bots [62].

Like other DL models, using RNNs requires two steps: (1) learning model weights through training, and (2) applying the model to predict the results of new requests, which is referred to as *serving*, or equivalently, inferencing or scoring. Training is a throughput-oriented task: existing systems batch the computation of multiple training inputs to obtain massive parallelism, leveraging GPUs to obtain high throughput. Users can often tolerate fairly long training time of hours and days because it is offline. Serving, on the other hand, makes online prediction of incoming requests, imposing different goals and unique challenges, which is the focus of this paper.

Latency and efficiency are the two most important metrics for serving. Interactive services often require responses to be returned within *a few or tens of milliseconds* because delayed responses could degrade user satisfaction and affect revenue [27]. Moreover, large-scale services handle massive request volumes and could require thousands of machines to serve a single model. Many RNN models from production services such as web search, advertisement, and conversational bots require intensive computation and could not be shipped because of serving latency violation and cost constraints.

Detailed investigation shows that popular DL frameworks, e.g., TensorFlow and CNTK, exhibit poor performance when serving RNNs. Consider the performance metric of floating point operations per second (flops), which is a standard measure for computations like DL that are dominated by floating-point calculations. Our test results show that on a modern Intel CPU with peak performance of 1.69Tflops, using TensorFlow/CNTK for RNN serving only gets less than 2% of hardware peak. This naturally raises many questions: Why is there such a big performance gap between hardware peak and the existing implementations? Are we dealing with an intrinsically challenging workload or less optimized systems? Would different hardware, such as GPU, help?

We carefully characterize RNN performance and answer the above questions.

First, RNN serving is an intrinsically challenging workload. Due to stringent latency SLA, online serving systems often process each request upon its arrival, or at best, batch a few requests whenever possible. With a batch size of 1 (or a few), the computation is dominated

*Both authors contributed equally. Order of appearance is random.

by several vector-matrix multiplications (or matrix multiplications), that have poor *data reuse* and thus are bottlenecked on cache/memory bandwidth. Since the speed of data transfer is far slower than the computational speed of CPUs, this leaves cores waiting for data instead of conducting useful computation, leading to poor performance and latency.

Second, existing DL frameworks rely on parallel-GEMM (GEneral Matrix to Matrix Multiplication), implementations which are not targeted to optimize the type of matrix multiplications (MMs) in RNN computations. parallel-GEMM is designed to optimize large MMs with high data reuse by hiding the data movement cost with ample computation [29]. MMs in RNNs are usually much smaller, fitting entirely in shared $L3$ cache, but with minimal data reuse: data movement from shared $L3$ cache to private $L2$ cache is the main bottleneck. Due to limited data reuse, parallel-GEMM can no longer hide the data movement, requiring different considerations and new techniques. Furthermore, as weights are repeatedly used at MMs of each step along the sequence, it presents a potential reuse opportunity from RNN domain knowledge, which parallel-GEMM does not exploit.

Lastly, would GPU help? RNN serving is computationally intensive but with limited parallelism. In particular, the amount of computation grows linearly with the sequence length: the longer the sequence, the more steps the computation carries. However, the sequential dependencies make it hard to parallelize across steps. As the batch size is also small in serving scenario, there is rather limited parallelism for RNN serving. As GPUs use a large number of relatively slow cores, they are not good candidates because most of the cores would be idle under limited parallelism; CPUs are a better fit with a smaller number but faster cores.

With the challenges and opportunities in mind, we develop novel techniques to optimize data reuse. We build DeepCPU, an efficient RNN serving library on CPUs, incorporating the optimization techniques. Our key techniques include (1) private-cache-aware partitioning, that provides a principled method to optimize the data movement between the shared $L3$ cache to private $L2$ cache with formal analysis; (2) weight-centric streamlining, that moves computation to where weights are stored to maximize data reuse across multiple steps of RNN execution. Both help overcome the limitation of directly applying parallel-GEMM and optimize data reuse on multi-core systems. We also leverage existing techniques, such as MM fusion and reuse-aware parallelism decision, in the new context of RNN optimization.

Effectively integrating these techniques together is non-trivial, requiring to search a large space to find optimized schedules. We model RNN computation using a Directed Acyclic Graph of Matrix Multiplication nodes (MM-DAG), supporting a rich set of optimization knobs such as partitioning (splitting a node) and fusion (merging nodes). It is well known that the traditional DAG scheduling problem of minimizing execution time by deciding the execution order of the nodes is NP-hard even in the absence of additional knobs [28]. The optimization knobs further enlarge the search space exponentially, and it is infeasible to exhaustively enumerate all schedules. We develop an efficient search strategy that requires far fewer calibration runs.

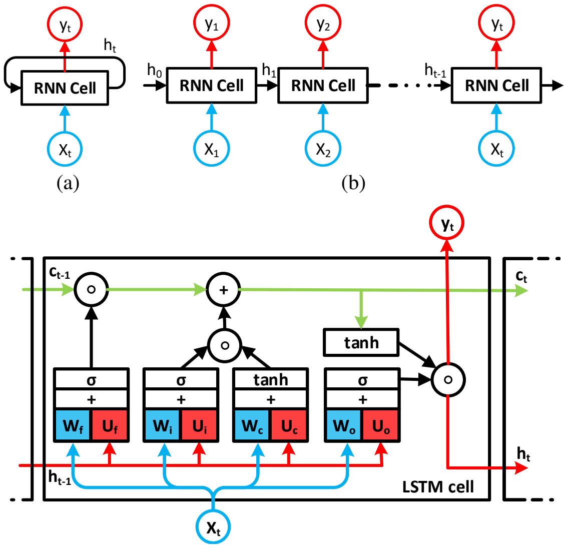
We compare DeepCPU with popular state-of-the-art DL frameworks, including TensorFlow and CNTK, for a wide range of RNN models and settings. The results show DeepCPU consistently outperforms them on CPUs, improving latency by an order of magnitude. DeepCPU also empowers CPUs to beat highly optimized implementations on GPUs. We further demonstrate its impact on three real-world applications. DeepCPU reduces their latency by 10–20 times in comparison to TensorFlow. To meet latency SLA, DeepCPU improves the throughput of the text similarity model by more than 60 times, serving the same load using less than 2% of machines needed by the existing frameworks.

The key contributions of the work include: 1) Characterizing performance limitations of the existing methods (Section 3). 2) Developing novel techniques and a search strategy to optimize data reuse (Section 4 and 5). 3) Building DeepCPU, a fast and efficient serving library on CPUs (Section 4 and 5). 4) Evaluating DeepCPU and showing order of magnitude latency and efficiency improvement against the existing systems (Section 6).

DeepCPU has been extensively used in the production of Microsoft to reduce serving latency and cost. It transforms the status of many DL models from impossible to ship due to violation of latency SLA to well-fitting SLA requirements. It empowers bigger and more advanced models, improving accuracy and relevance of applications. DeepCPU also greatly improves serving efficiency, saving thousands of machines and millions of dollars per year for our large-scale model deployments.

2. Background

An RNN models the relationships along a sequence by tracking states between its steps. At each step t (Fig. 1a), it takes one unit of input x_t (e.g., a token in a text, or a phoneme in a speech stream) and makes a prediction y_t based on both the current input x_t and the previous hidden (or cell) state h_{t-1}. The hidden states $\{h_t\}$ form a loop, allowing information to be passed from one step to the next. The block of computation per step is called an RNN cell, and the same cell computation is used for all inputs of the sequence. An RNN (sequence) computation can be viewed as an unrolled chain of cells (Fig. 1b).



(a) (b)

(c)

Figure 1: a) RNN with a recurrent structure. b) Unrolled RNN. c) LSTM structure.

LSTM/GRU. There are many variations of RNNs, inheriting the recurrent structure as above but using different cell computations. The two most popular ones are Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) network, best known for effectively catching long-term dependencies along sequences. We use LSTM as an example and illustrate its cell computation:

$$\begin{aligned}
i_t &= \sigma(\mathbf{W}_i \cdot x_t + \mathbf{U}_i \cdot h_{t-1} + b_i) \\
f_t &= \sigma(\mathbf{W}_f \cdot x_t + \mathbf{U}_f \cdot h_{t-1} + b_f) \\
o_t &= \sigma(\mathbf{W}_o \cdot x_t + \mathbf{U}_o \cdot h_{t-1} + b_o) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tanh(\mathbf{W}_c \cdot x_t + \mathbf{U}_c \cdot h_{t-1} + b_c) \\
h_t &= o_t \circ \tanh(c_t) \ .
\end{aligned}$$

Here $\sigma(\cdot)$ denotes the sigmoid function. Online tutorials [7, 12] describe good insights of the formulation on how it facilitates learning. Here we focus on describing the main computations. We denote E as the input dimension of the input vector x_t, and H as the hidden dimension of the hidden vector h_t. LSTM includes 4 input MMs, which multiply input vector x_t with four input weight matrices $\mathbf{W}_{\{i,f,o,c\}}$ of size $E \times H$ each (marked as blue in Fig. 1c). It has 4 hidden MMs, which multiply hidden vector h_{t-1} with four hidden weight matrices $\mathbf{U}_{\{i,f,o,c\}}$ of size $H \times H$ each (red in Fig. 1c). Within each cell, there is no dependency among the 8 MMs, and across cells, the hidden state of step t depends on step $t-1$ (as shown by Fig. 1c). LSTM also consists of a few element-wise additions ($+$) and products ($\circ$), as well as activation functions such as σ and tanh.

Similar to the LSTM cell, GRU cell has 6 instead of 8 MMs but with additional dependencies within them [22].

Single vs. batch mode. To make real-time predictions, online requests are often processed one by one as they arrive, or occasionally, under a small batch. Given a batch size of B, the batched input x_t can be represented as a matrix of size $B \times E$, which transforms the underlying computation from a vector-matrix to a matrix-matrix multiplication, exposing more opportunities for data reuse. However, because of tight latency requirements and spontaneous request arrivals, the batch size at serving is usually much smaller (e.g., 1 to 10) than the large mini-batch size (often hundreds) during training.

3. Performance Characterization

Existing DL frameworks such as TensorFlow/CNTK implement RNNs as a loop of cell computation: as shown in Lis. 1, 8 MMs in the LSTM cell are fused into a single MM, executed using parallel BLAS libraries such as Intel-MKL [3], OpenBLAS [6] or Eigen [1]. We measure their performance in serving scenarios with small batch size from 1 to 10. On a dual-socket Xeon E5-2650 CPU machine, we often observe performance of < 30Gflops: less than 2% of the machine peak of 1.69Tflops. What is the cause of such a big gap?

Listing 1: LSTM Implementation in TensorFlow/CNTK

```
for t in input_sequence:
    $[f'_t \;\; i'_t \;\; o'_t \;\; c'_t] = [x_t \;\; h_{t-1}] \begin{bmatrix} \mathbf{W}_f & \mathbf{W}_i & \mathbf{W}_o & \mathbf{W}_c \\ \mathbf{U}_f & \mathbf{U}_i & \mathbf{U}_o & \mathbf{U}_c \end{bmatrix}$
    $c_t = \sigma(f'_t) \circ c_{t-1} + \sigma(i'_t) \circ \tanh(c'_t)$
    $h_t = \sigma(o'_t) \circ \tanh(c_t)$
```

The first step of performance analysis is to identify the dominating computation. In RNNs, the total amount of computation is dominated by MMs. The total ops in MMs per RNN cell are $\mathcal{O}(B \times (E+H) \times H)$, and the total ops in element-wise operations and activations functions are $\mathcal{O}(B \times H)$. Typically the total number of ops in MMs is two to three orders of magnitude larger than the rest combined. As such, RNN performance primarily depends on the MMs, which is the focus of this study.

We analyzed MMs in the RNNs, and identified three key factors causing poor performance.

i) Poor data reuse. Data reuse at a particular level of memory hierarchy is a measure of the number of computational ops that can be executed per data load/store at that level of memory hierarchy. Assuming a complete overlap between computation and data movement (best case scenario), the execution time of a computation can be estimated as a function of the data reuse using the roofline model [65] as

$$\begin{aligned}
Time &\geq Max(DataMoveTime, CompTime) \quad (1) \\
&= Max(\frac{DataMoved}{DataBandwidth}, \frac{TotalComp}{Peak}) \\
&= Max(\frac{TotalComp/Reuse}{DataBandwidth}, \frac{TotalComp}{Peak})
\end{aligned}$$

Based on this execution time, note that poor data reuse results in poor performance because on modern architectures, the computational throughput is significantly higher than the data movement throughput. Let us look at an example of $L3$ to $L2$ bandwidth since all RNN models we have seen fit in $L3$ cache of modern CPUs: the *peak* computational performance of a Xeon E5-2650 machine is 1.69Tflops while the observable *DataBandwidth* between $L3$ and $L2$ cache on it is 62.5 GigaFloats/s (250 GB/s), measured using the stream benchmark [8]. If the reuse is low, the total execution time is dominated by the data movement, resulting in poor performance.

This is indeed the case for RNN in serving scenario where the batch size tends to be very small. To see this, consider an MM:$\mathbf{C}[i,j] = \sum_k \mathbf{A}[i,k] \times \mathbf{B}[k,j]$. If we assume that both the inputs and the outputs reside in $L3$ cache at the beginning of the computation, then both the inputs and the outputs must be read from $L3$ cache to $L2$ cache at least once, and the outputs must be stored from $L2$ cache to $L3$ cache at least once during the MM. Therefore, the maximum possible data reuse during this MM from $L2$ cache is given by $\frac{2 \times I \times J \times K}{|\mathbf{A}|+|\mathbf{B}|+2|\mathbf{C}|}$, where I, J and K are the size of indices i, j and k. Similarly, the fused MM of LSTM has the shape $[B, E+H] \times [E+H, 4H]$, and its data reuse is:

$$MaxDataReuse = \frac{8 \times B \times H \times (E+H)}{|Input|+|Weights|+2|Output|} \quad (2)$$

$$= \frac{8 \times B \times H \times (E+H)}{B \times (E+H) + 4 \times (E+H) \times H + 8 \times B \times H} \quad (3)$$

When batch size $B \ll \min(H,E)$, the maximum data reuse in Eqn. 2 reduces to $2B$. Take $B = 1$ as an example: the best achievable performance of LSTM on the Xeon E5-2650 machine is at most 125Gflops based on the measured $L3$ bandwidth of 250 GB/s. This is less than 8% percent of the machine's peak of 1.69Tflops.

ii) Sub-optimal MM partitioning. Parallel-GEMM libraries are designed to optimize performance of large MMs that have significant data reuse ($>$ 1000). They exploit this reuse from $L2$ cache level using loop-tiling to hide the data movement cost from both memory and $L3$ cache [29]. In contrast, the amount of reuse in RNNs is in the order of B, which is often a small value between 1 and 10 for most serving cases. This is not enough to hide the data movement cost even though MMs in RNN are small enough to fit in $L3$ cache. In the absence of large reuse, the performance of parallel-GEMM is limited by the data movement cost between shared $L3$ cache and private $L2$ caches. Parallel-GEMM is sub-optimal at minimizing this data movement.

More specifically, $L3$ cache on a modern CPU feeds to *multiple* $L2$ caches that are private to each core. During RNN computations, some data might be required by multiple cores, causing multiple transfers of the same piece of data from $L3$ cache. Thus, the total data movement between $L3$ and $L2$ caches depends on the partitioning of the MM computation space and its mapping to the cores. For example, if we split an MM computation among two cores, such that the first core computes the upper half of the output matrix **C**, while the second core computes the lower half, then input matrix **B** must be replicated on $L2$ cache of both cores, as the entire matrix **B** is required to compute both halves of matrix **C**. Alternatively, if the computation is split horizontally, then the input matrix A must be replicated on $L2$ cache of both cores. Different partitionings clearly result in different amount of data reuse. Parallel-GEMM does not always produce a partitioning that maximizes this data reuse. Libraries specialized for small matrices are not sufficient either, as some focus only on sequential execution [56] while others focus on MM small enough to fit in $L1$ cache [43].

iii) No data reuse across the sequence. During serving, weight matrices of RNNs remain the same across the sequence, but existing solutions do not take advantage of that to optimize data reuse. More precisely, parallel-GEMM used to execute the MMs is not aware of this reuse across the sequence. During each step of the sequence, the weight matrix could be loaded from $L3$ cache to $L2$ cache. However, it is possible to improve performance of RNNs by exploiting this data reuse.

Beyond MM: Beyond limited data reuse at MMs, existing RNN implementations in DL frameworks such as TensorFlow have other performance limiting factors, e.g., data transfer overheads among operators, buffer management overheads, unoptimized activation functions, which we address in DeepCPU. For example, we develop efficient SIMD implementations of *tanh* and *sigmoid* activation functions using continued fraction expansion, supporting any desired degree of precision by adjusting the number of terms to terminate the expansion [60]. Since these improvements mostly require good engineering practice than novel methods, we did not discuss them in detail for the interest of space.

4. Challenges and Strategies

Challenges. Finding an optimized implementation for RNN execution that maximizes data reuse while also efficiently using low-level hardware resources (such as SIMD hardware) is challenging due to the explosive space of optimization knobs and execution schedules. Practically infinite number of valid choices can be obtained through loop permutations, loop fusions, loop unrolling, unroll factor selection, loop tiling, tile-size selection, MM reordering, vectorization, register tiling, register tile size selection, parallel loop selection, parallelization granularity selection, thread-to-core mapping etc., and their combinations. Furthermore, enabling those optimization knobs and creating a schedule generator for all choices is a non-trivial engineering task. Additionally, the optimal choice is dependent on both hardware architecture and RNN parameters: a single solution will not work for all cases and an optimized schedule needs to be tuned case by case in an efficient manner. All of the above make the problem challenging in practice.

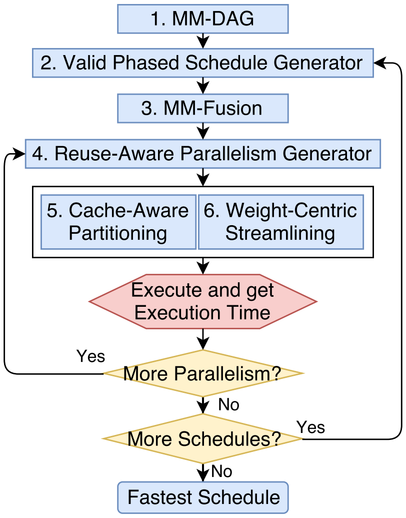

1. MM-DAG
2. Valid Phased Schedule Generator
3. MM-Fusion
4. Reuse-Aware Parallelism Generator
5. Cache-Aware Partitioning
6. Weight-Centric Streamlining
Execute and get Execution Time
Yes
More Parallelism?
No
More Schedules?
Yes
No
Fastest Schedule

Figure 2: DeepCPU optimization overview.

Strategies: DeepCPU overview. To overcome these challenges, we judiciously define the search space and identify the most important techniques to boost data locality. This empowers efficient search within a selective set of optimization knobs and schedules for obtaining the best RNN performance. We build the entire optimization pipeline into a library, which we call DeepCPU. Fig. 2 highlights its key features and workflow.

An important start of the optimization is to define a concise search space, which we develop upon two insights. (1) We identify the most performance critical operators, MMs, and model the computation graph connecting them to capture the first-order impact. We can do this by constructing a Matrix Multiplication Directed Acyclic Graph (MM-DAG) to represent the RNN computation, where each node represents an MM and edges represent dependencies among them. This model allows us to build schedules using MMs as the basic building blocks, capturing key computations while abstracting away other low-level details. (2) Instead of examining all valid schedules for the MM-DAG, which is not trackable, we can leverage the iterative nature and other properties of RNNs, prune search space to deduplicate the performance-equivalent schedules, and remove those that cannot be optimal. These two insights are implemented as $\boxed{1}$ and $\boxed{2}$ in Fig. 2.

We then identify and develop four techniques to effectively boost data locality for RNNs, applying them on each schedule (shown as $\boxed{3}$,$\boxed{4}$,$\boxed{5}$,$\boxed{6}$ in Fig. 2):

- **MM-fusion:** fuses smaller MMs into larger ones, improving data reuse;
- **Reuse-aware parallelism generator:** identifies best parallelism degree within and across MMs through auto-tuning, jointly considering locality;
- **Private-cache-aware-partitioning (PCP):** optimizes data movement between shared $L3$ cache and private $L2$ cache with a novel and principled partitioning method;
- **Weight centric streamlining (WCS):** maps the partitions produced by PCP to cores in a way that enables reuse of weights across the sequence.

The parallelism generator $\boxed{4}$ iterates over different choices on parallelism degrees. For a parallelism choice, we use PCP $\boxed{5}$ to obtain locality optimized parallel partitions. The partitions are then mapped to cores using WCS $\boxed{6}$. Individual partitions are implemented using highly optimized single-threaded BLAS library which optimizes for low-level hardware resources such as $L1$ cache and SIMD instruction set. DeepCPU applies this schedule to obtain the execution time, and loop over to find the best parallelism choice. Once this process is completed for all schedules generated by $\boxed{2}$, DeepCPU simply chooses the schedule that is the fastest. This calibration process is often called once during model construction, and then the optimized schedule is repeatedly used for serving user requests of the model.

In the design of DeepCPU, we deliberately combine analytical performance analysis (at search space pruning and PCP) with empirical calibration (to measure the combined impact of locality and parallelism). The former effectively reduces the search space, saving tuning time to run many suboptimal/redundant schedules. The latter reliably measures the actual execution time to capture complex software and hardware interaction, which can hardly be accurately estimated. This combination empowers both effectiveness and efficiency.

5. DeepCPU Optimizations

This section dives into DeepCPU optimizations from refining search space to locality optimizations. We conclude it by demonstrating the performance breakdown and impact of these optimizations.

5.1. MM-DAG Scheduling

DeepCPU models RNN computations as MM-DAGs and optimizes the schedules to execute them. Given an MM-DAG, a valid schedule determines an execution ordering of its nodes that satisfies all the dependencies. We consider only those valid schedules that are composed of phases: A *phased schedule* executes an MM-DAG in a sequence of phases $S_1, S_2, S_3, ..., S_i, ...$, where each phase S_i represents a non-overlapping subset of nodes and $S = \sum_i S_i$ consists of all nodes. There is a total ordering between phases such that if $i < j$, then all nodes in S_i must be executed before S_j. However, nodes within a phase can be executed in parallel. Lst. 2 shows two examples of valid phased schedules for LSTM. In Schedule 1, all MMs at a timestep t are in Phase t.

The phases can be divided into two categories: i) If a phase consists of an MM that has dependency across the timesteps, we call it a *time-dependent phase*, e.g., those MMs taking hidden state h_t as inputs, ii) Otherwise, if a phase does not contain any MM that has dependency across the sequence, we call it a *time-independent phase*. For example, in Schedule 2 of Lst. 2, Phase 1 is time-independent, and consists of all the MMs computing input transformation (with weights $\mathbf{W_i}, \mathbf{W_f}, \mathbf{W_c}$ and $\mathbf{W_o}$) across all timesteps; all other phases are time-dependent, requiring the value of h_{t-1} to compute h_t.

Listing 2: Phased LSTM Schedule-1 and 2

```
//Phased LSTM Schedule 1
for t:
  Phase t:   //time-dependent
    $\mathbf{W}_i \cdot x_t, \mathbf{W}_f \cdot x_t, \mathbf{W}_c \cdot x_t, \mathbf{W}_o \cdot x_t$
    $\mathbf{U}_i \cdot h_{t-1}, \mathbf{U}_f \cdot h_{t-1}, \mathbf{U}_c \cdot h_{t-1}, \mathbf{U}_o \cdot h_{t-1}$

//Phased LSTM Schedule 2
Phase 1:   //time-independent
   $\mathbf{W}_i \cdot x_0, .., \mathbf{W}_i \cdot x_t, \mathbf{W}_f \cdot x_0, .., \mathbf{W}_f \cdot x_t,$
   $\mathbf{W}_c \cdot x_0, .., \mathbf{W}_c \cdot x_t, \mathbf{W}_o \cdot x_0, .., \mathbf{W}_o \cdot x_t$
for t:
  Phase (t+1):   //time-dependent
     $\mathbf{U}_i \cdot h_{t-1}, \mathbf{U}_f \cdot h_{t-1}, \mathbf{U}_c \cdot h_{t-1}, \mathbf{U}_o \cdot h_{t-1}$
```

Reducing search space. We propose three rules to prune the search space, removing sub-optimal and redundant schedules: i) Time-dependent phases must have symmetry across timesteps. As RNN computation is identical across timesteps, the fastest schedule for executing each timestep should also be identical. ii) If two consecutive phases are of the same type, then there must be a dependency between the two phases. If no dependency exists then this schedule is equivalent to another schedule where a single phase consists all MMs in both phases. iii) We compute time-independent phases before all dependent ones, as shown in Schedule 2 of Lst. 2. Having phases of the same type in consecutive order increases reuse of weights.

5.2. Data Locality Optimizations

DeepCPU improves data reuse within each phase and across phases through four techniques.

5.2.1 Fusion of MMs

DeepCPU fuses all possible MMs within each phase — *Two MMs can be fused into a single MM if they share a common input matrix.*

How to fuse? Consider two MMs, $MM1 : \mathbf{C1}[i1, j1] = \sum_{k1} \mathbf{A1}[i1,k1] \times \mathbf{B1}[k1, j1]$ and $MM2 : \mathbf{C2}[i2, j2] = \sum_{k2} \mathbf{A2}[i2,k2] \times \mathbf{B2}[k2, j2]$. W.l.o.g., assume $\mathbf{A1}[i1,k1] = \mathbf{A2}[i1,k1]$, as shared input matrix. The two MMs can be fused into a single one $MM12$ by concatenating $\mathbf{B1}$ and $\mathbf{B2}$, and $\mathbf{C1}$ and $\mathbf{C2}$ along the column, i.e., $\mathbf{C12}[i1, j12] = \sum_{k1} \mathbf{A1}[i1,k1] \times \mathbf{B12}[k1, j12]$ where $\mathbf{B12}[k1, j1] = \mathbf{B1}[k1, j1]$, $\mathbf{B12}[k2, J1 + j2] = \mathbf{B2}[k2, j2]$, and $\mathbf{C12}[i1, j1] = \mathbf{C1}[i1, j1]$, $\mathbf{C12}[i2, J1 + j2] = \mathbf{C2}[i2, j2]$ ($J1$ is the size of index $j1$).

Why fuse? Fusion improves data reuse. Consider using any GEMM implementation to execute $MM1$ and $MM2$ without fusion. While both $MM1$ and $MM2$ share a common input, GEMM is not aware of this reuse and could not take advantage of it. However, if we fuse them, this reuse is explicit in the MM and GEMM can exploit it to improve both performance and scalability.

5.2.2 Reuse-aware Parallelism Generator

Parallelism boosts compute capacity but may also increase data movement. This part discusses the relation of locality and parallelism, and our parallelism strategy.

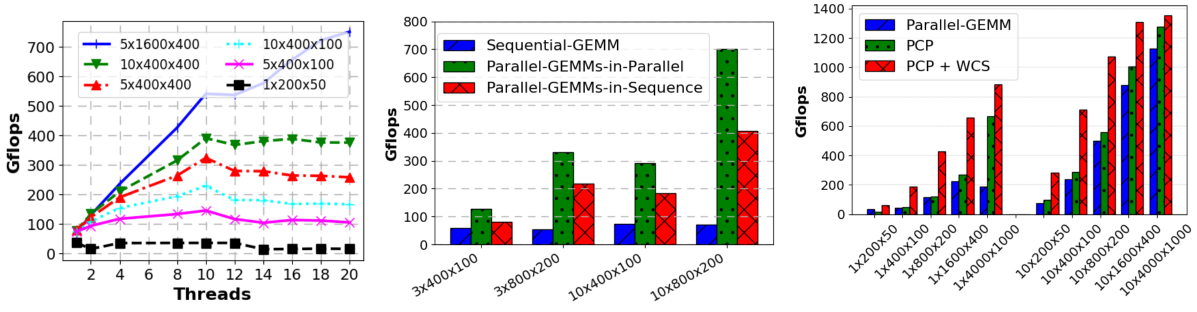
How to parallelize a single MM? Executing an MM with the maximum available parallelism is not always the best option for performance. As the parallelism increases, either the input or output must be replicated across multiple $L2$ private caches, thus increasing the total data movement. Once the level of parallelism reaches a certain threshold, the performance is limited by the data movement instead of the computational throughput. As shown in Fig. 3a, the MM performance degrades after certain parallelism. It is crucial to find the optimal level of parallelism instead of applying the common wisdom of using all available cores.

How to parallelize concurrent MMs? Multiple MMs within a phase do not have any dependencies. DeepCPU executes them as *Parallel-GEMMs-in-Parallel*, where multiple MMs are executed concurrently with each MM executing in parallel. For example, to compute two independent MMs, $M1$ and $M2$, on P cores, we run $M1$ and $M2$ in parallel, each using $P/2$ cores. This is in contrast with Parallel-GEMMs-in-Sequence, where we run $M1$ first using P cores followed by $M2$. Parallelizing an MM across multiple cores increases the data movement from $L3$ to $L2$ cache. In contrast, executing multiple MMs in parallel across multiple divided groups of cores allows each group to work on a unique MM without requiring data replication across them, improving data reuse while maintaining the same parallelism level. Fig. 3b shows empirical results. We run two independent and identical MMs with increased parallelism and report the best performance achieved. Parallel-GEMMs-in-Parallel significantly outperforms Parallel-GEMMs-in-Sequence.

How to optimize parallelism degree? Finding the optimal parallelism degree analytically is non-trivial as it depends on many architectural parameters. However, it is also not necessary in practice. DeepCPU applies Parallel-GEMMs-in-Parallel if a phase has multiple fused MMs. It then uses auto-tuning to identify the optimal parallelism for the phase quickly, as the number of cores on a modern multi-core CPU is less than two orders of magnitude and well-known RNN operators such as LSTMs/GRUs have at most two fused MMs per phase.

5.2.3 Private-Cache-Aware Partitioning (PCP)

We develop PCP, a novel private-cache-aware partitioning strategy for executing MMs across multicores to optimize $L2$ reuse within and across phases. PCP provides a principled method to optimize data movement with for-



(a) (b) (c)

Figure 3: For MMs of different sizes, (a) shows performance results by increasing the parallelism degree. (b) ompares Parallel-GEMMs-in-Parallel vs Parallel-GEMMs-in-Sequence, plus Sequential-GEMM as baseline. C) compares parallel-GEMM, PCP with and without weight-centric streamlining for running MMs.

mal analysis: For a given MM with parallelism degree P, we show PCP produces a P-partitioning of the computation space such that the total data movement between $L3$ and $L2$ cache is minimized. DeepCPU employs PCP to generate a locality-optimized schedule for each parallelism configuration without requiring to empirically calibrate different partitions and measure their performance.

Reuse within phases. Suppose an MM $\mathbf{C}[i,j] = \sum_k \mathbf{A}[i,k] \times \mathbf{B}[k,j]$ has P partitions, where X_i, X_j and X_k are the number of partitions along each of the i, j, k dimensions and $X_i \times X_j \times X_k = P$. We first derive the total data movement between $L3$ and $L2$ cache as a function of the partitions. This data movement depends on the relation between the size of the input and output matrices of the MM and the sizes of the $L3$ and $L2$ caches. For all RNNs of interest in serving scenario, we observe that the input matrix is much smaller than $L2$ cache, and the sum of all matrices fit in $L3$ cache. Under such conditions, we prove in Lemma 5.1 and Theorem 5.2 that the total data movement between $L3$ and $L2$ cache is equal to $X_j|\mathbf{A}| + X_i|\mathbf{B}| + 2X_k|\mathbf{C}|$. By choosing X_i, X_j, and X_k that minimizes this quantity, PCP obtains a parallel partitioning that maximizes data reuse from $L2$ cache.

Lemma 5.1. *The tight bound on data movement between a slow memory and a fast memory of size S for an MM* $\mathbf{C}[i,j] += \sum_k \mathbf{A}[i,k] \times \mathbf{B}[k,j]$ *is given by* $|\mathbf{A}| + |\mathbf{B}| + 2|\mathbf{C}|$, *when* $S \geq min(|\mathbf{A}|, |\mathbf{B}|, |\mathbf{C}|) + H + 1$. *Here we assume that the inputs and output matrices initially reside in the slow memory, and the final output must also reside in the slow memory. H is a constant not greater than* $\max(I,J,K)$, *where I, J and K are the sizes of indices i, j and k respectively.*

Proof. Lower bound: As both inputs and outputs originally reside in slow memory they must be read to fast memory at least once to compute the MM. After computation, the output must be written to slow memory at least once. That gives $|\mathbf{A}| + |\mathbf{B}| + 2|\mathbf{C}|$ as a lower bound.

Upper bound: W.l.o.g., assume $\mathbf{A}$ fits in S. Lst. 3 shows a schedule where the total data movement between slow and fast memory is given by $|\mathbf{A}| + |\mathbf{B}| + 2|\mathbf{C}|$, when $S \geq |\mathbf{A}| + K + 1$. Note H (=K) is a (small) buffer space to hold a single column of $\mathbf{B}$ during the computation. □

Listing 3: MM Schedule that achieves data movement of $|\mathbf{A}| + |\mathbf{B}| + 2|\mathbf{C}|$ when $S \geq |\mathbf{A}| + K + 1$

```
//C[i,j] = Σk A[i,k] × B[k,j]
Load A[*,*] in A_buf //MemReq = |A|
for j
  Load B[*,j] in B_buf //MemReq = K
  for i
    Load C[i,j] in c //MemReq  = 1
    for k
      c += A_buf[i,k] × B_buf[k]
    Store c in C[i,j]
```

Theorem 5.2. *Consider P cores on a CPU, and an MM* $\mathbf{C}[i,j] += \sum_k \mathbf{A}[i,k] \times \mathbf{B}[k,j]$, *where* $|\mathbf{A}| + |\mathbf{B}| + |\mathbf{C}| \leq |L3Cache|$ *and* $min(|\mathbf{A}|, |\mathbf{B}|, |\mathbf{C}|) + H + 1 \leq |L2Cache|$. *H is a constant not greater than* $max(I,J,K)$, *where I, J and K are the sizes of indices i, j and k. For a P-way partitioning* $\langle X_i, X_j, X_k \rangle$ *where* $X_i \times X_j \times X_k = P$, *a tight bound on the data movement between L3 and L2 cache is given by* $X_j|\mathbf{A}| + X_i|\mathbf{B}| + 2X_k|\mathbf{C}|$.

Proof. Each of the partitions given by $\langle X_i, X_j, X_k \rangle$ is an MM of size $\frac{I}{X_i} \times \frac{J}{X_j} \times \frac{K}{X_k}$. From Lemma. 5.1 we see that a tight bound on the data movement between $L3$ cache and $L2$ cache for each of these sub-MMs is given by $\frac{I \times K}{X_i \times X_k} + \frac{K \times J}{X_k \times X_j} + 2\frac{I \times J}{X_i \times X_j}$. Thus the total data movement for all partitions is given by $X_i \times X_j \times X_k \times (\frac{I \times K}{X_i \times X_k} + \frac{K \times J}{X_k \times X_j} + 2\frac{I \times J}{X_i \times X_j}) = X_j|\mathbf{A}| + X_i|\mathbf{B}| + 2X_k|\mathbf{C}|$. □

Reuse across phases. PCP so far maximizes the data reuse by considering each phase independently. However, identical time-dependent phases (TDPs) across a sequence have data reuse between them. For each MM in these phases, weight matrices stay the same. We extend PCP to exploit the reuse in weights across phases.

For a given P-partitioning strategy $\langle X_i, X_j, X_k \rangle$, the weight matrix $\mathbf{B}$ is divided into blocks of size $\frac{|\mathbf{B}|}{X_j \times X_k}$. If this block fits in $L2$ cache of an individual core, then it will not be evicted from $L2$ cache for the entire computation sequence as long as the mapping between the MM partitions and the compute cores does not change. In such cases, denoting the sequence length of RNN as *seq_len*, the total data movement is given by

$$seq\_len \times (X_j|\mathbf{A}| + 2X_k|\mathbf{C}|) + X_i|\mathbf{B}|$$

as the weight matrix **B** needs to be read only once from $L3$ cache at the first time step. In general, total data-movement between $L3$ and $L2$ caches is calculated as

$$\begin{cases} seq\_len \times (X_j|\mathbf{A}| + 2X_k|\mathbf{C}|) + X_i|\mathbf{B}| & \text{if } \frac{|\mathbf{B}|}{X_j * X_k} \leq |L2| \\ seq\_len \times (X_j|\mathbf{A}| + X_i|\mathbf{B}| + 2X_k|\mathbf{C}|) & \text{if } \frac{|\mathbf{B}|}{X_j * X_k} > |L2| \end{cases}$$

By minimizing this piecewise function, we maximize the data reuse across a sequence. In practice, it is not necessary for a block of the weight matrices to fit entirely in $L2$ cache. As long as the block is not much larger than $L2$ cache, we can still get partial reuse.

5.2.4 Weight-Centric Streamlining (WCS)

WCS is our implementation to enable full-fledged PCP, supporting reuse of weight matrices across TDPs. For a given parallelism degree, PCP produces a partitioning such that the weights required to compute the partition fit in $L2$ cache of a single core (when possible), allowing the weights to be reused from $L2$ cache across TDPs, without being evicted. However, to ensure this reuse, the computation must be conducted at where the weights are, i.e., the mapping between parallel partitions and the cores that execute them must not change across TDPs.

To this end, we use OpenMP [24] to create a parallel region that spans the entire RNN sequence of computation. The parallelism degree is equal to the max parallelism degree among all phases in the schedule. Each thread in the parallel region is responsible for executing at most a single parallel partition during each phase. Some threads may remain idle during phases where the parallelism degree is less than the number of threads. Each thread ID is mapped to a unique partition ID, and this mapping is identical across TDPs. In essence, we alternate the order of the sequence loop and the parallel region such that the sequence loop is inside the parallel region, shown as ParallelOuterRNN in Lst. 4.

Listing 4: Parallel Outer vs Parallel Inner RNN

```
ParallelOuterRNN(intput_sequence, output)
  #pragma omp parallel
    int id = omp_get_thread_num()
    for t in intput_sequence:
      ComputeRNNOuterParallel(id, t, output)

ParallelInnerRNN(intput_sequence,output)
  for t in intput_sequence:
    #pragma omp parallel
      int id = omp_get_thread_num()
      ComputeRNNInnerParallel(id, t, output)
```

This has two major advantages over creating parallel regions inside the sequence loop as ParallelInnerRNN, i) it allows easy pinning of each MM parititon to a particular core across RNN steps. In OpenMP, threads in each parallel region have their local thread IDs starting from 0. A unique mapping between this local thread ID and the global thread ID is not guaranteed across multiple parallel regions separated in time. Thread affinity settings allow binding global thread IDs to cores or hyperthreads, but not local thread IDs. By creating a single parallel region, we create a unique mapping between a local thread ID and the global thread ID throughout the computation, which ensures that an MM partition is always executed on the same core across the entire sequence. ii) It reduces the overhead of creating parallel regions. Instead of opening and closing parallel regions during each step of the RNN sequence, we only create a parallel region once for the entire computation.

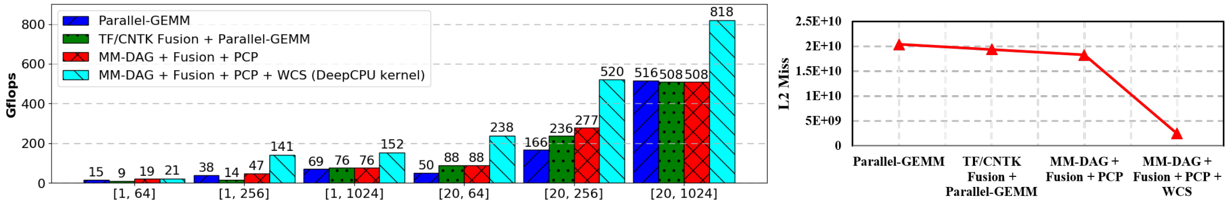
Fig. 3c compares performance of running a sequence of parallel-GEMM and PCP with/without WCS for varied sizes of MMs. The latter two consistently outperform the former, but the full benefit of PCP (across phases) is realized only when used together with WCS.

5.3. Performance Impact of Optimization Techniques

We compare four implementations using different LSTM configurations: i) Parallel-GEMM(baseline): Runs each step of LSTM as 8 MMs in sequence, and each MM is executed with Intel-MKL parallel-GEMM. ii) TensorFlow/CNTK Fusion: the fused MM (as Lst. 1) is executed using Intel-MKL parallel-GEMM. iii) MM-DAG+Fusion+PCP: All optimizations in DeepCPU except WCS. iv) DeepCPU Kernel: All aforementioned optimizations.

Results. TensorFlow/CNTK Fusion has roughly the same performance as baseline. MM-DAG+Fusion+PCP is as good as or better than both of them. It searches for the fused phased schedules including TensorFlow/C-NTK fusion, as well as those that increase reuse by fusing across time steps. It also applies PCP for better partitioning. However, it does not ensure that MMs sharing same weights are mapped to the same core. In contrast, DeepCPU kernel is often much faster, particularly for small batch sizes where the reuse is small within a single phase and reuse across TDPs must be exploited for better performance. Even for larger batch size with the input/hidden dimension 256 and 1024, where the total size of the weight matrices is larger than the $L2$ cache but individual weight blocks fit in $L2$ cache, DeepCPU kernel offers good speedup by enabling reuse of weights across TDPs.

Performance counters. We measure the amount of data movement from $L2$ to $L3$ through $L2$ cache misses using *L2_RQSTS.ALL_DEMAND_MISS* counter in Intel® VTune™ Amplifier [2]. Fig.4b shows DeepCPU significantly reduces $L2$ cache misses (by 8 times), verifying its effectiveness on locality optimization.



(a) (b)

Figure 4: (a) Performance of LSTMs (in the form of [batch size, input/hidden dimension]) with different optimization techniques. (b) $L2$ cache misses for config [20, 256]. Measured at $sequence\_length = 100$ with 2000 iterations.

Search space size. DeepCPU finds the optimal execution schedule with just a few hundred calibration runs. In the example of LSTM, we search approximately $P \times Q$ configurations by generating $P = \#cores$ parallelism choices, and Q phased schedules that satisfies all three pruning criteria described in Sec. 5.1. For LSTMs, $Q < 20$, which can be verified by enumerating all such valid schedules. Per parallelism choice, PCP identifies optimized partitioning analytically (e.g., integer programming) without requiring additional empirical exploration, greatly saving search space. This search/calibration process is often called once during model construction, and then the optimized schedule is repeatedly used for serving upcoming user requests.

6. Evaluation

We compare DeepCPU with RNN implementations from state-of-the-art DL frameworks: DeepCPU is an order of magnitude faster than TensorFlow and CNTK for a wide range of RNN models and configurations on CPUs. DeepCPU also outperforms GPU implementations significantly for most of the cases. Furthermore, we test DeepCPU on real-world applications used in production: it transforms these models from impossible to ship due to latency violation to well-fitting SLA while also saving millions of dollars in infrastructure cost.

Hardware. Our evaluation is conducted on a server with two 2.20 GHz Intel Xeon E5-2650 V4 processors, each of which has 12-core (24 cores in total) with 128GB RAM, running 64-bit Linux Ubuntu 16.04. The peak Gflops of the CPU is around 1.69Tflops. The server has one Nvidia GeForce GTX TITAN X which is used for measuring RNN performance on GPU.

6.1. RNN Performance Comparison

Workload. We evaluate LSTM/GRU by varying input dimension, hidden dimension, batch size, and input sequence length to cover a wide range of configurations.

Comparison frameworks. There are many DL frameworks such as TensorFlow [13], CNTK [52], Caffe2 [37], Torch [41], Theano [17], and MXNet [19] that support RNNs on CPUs and GPUs. We compare DeepCPU with TensorFlow and CNTK. We choose TensorFlow because it is adopted widely. We use TensorFlow version 1.1.0 with Accelerated Linear Algebra (XLA) compiler, optimizing pointwise kernels, and with Intel Math Kernel Library (MKL) for efficient matrix operations. We let TensorFlow pick appropriate degrees for inter-op and intra-op parallelism. We choose CNTK since a recent study showed that it achieves good performance on RNNs [55]. CNTK also uses MKL and sets the number of threads equal to the number of cores for MMs by default. On GPUs, we evaluate TensorFlow, CNTK and a highly optimized cuDNN implementation [14, 20].

Speedup on CPUs. Table 1 presents the execution time and speedup results of DeepCPU, in comparison to TensoFlow and CNTK on CPUs, covering a wide range of RNN model sizes. The first four columns describe the specification of RNNs: input dimension, hidden dimension, batch size, and sequence length. Both absolute execution time and speedup are reported. Speedup is measured as the ratio between the execution times of TensorFlow (or CNTK) versus DeepCPU, e.g., a value of 2 indicates that DeepCPU is 2 times faster. To make reliable measurement, we run each config 2000 times and report the average. The results show that *DeepCPU significantly and consistently outperforms TensorFlow and CNTK, with speedup in the range of 3.7 to 93 times, and average speedup of 18X among all tested configurations.*

Next, we conduct an in-depth performance comparison, showing how model parameters affect the results, on both CPU and GPU, across 6 implementations.

Varying input/hidden dimension. Fig. 5a reports the execution time and Gflops of LSTMs with varying input/hidden dimension from 32 to 1024. This is the range of dimension size commonly observed from RNN models in practice. Here we choose batch size of 1 to represent a common case in serving. As expected, the execution time for all implementations increases with the increase in dimension size. However, compared to all implementations, DeepCPU always has the shortest execution time and the highest Gflops on both CPUs and GPUs for all sizes. Note that the y-axis of the execution time is in log-scale, so the actual gap is larger than it appears. DeepCPU is more than an order of magnitude faster than both TensorFlow and CNTK on CPUs. On GPUs, DeepCPU has significantly higher performance when the dimension size is small or medium (e.g., less than 256). As the dimension size gets larger, this perfor-

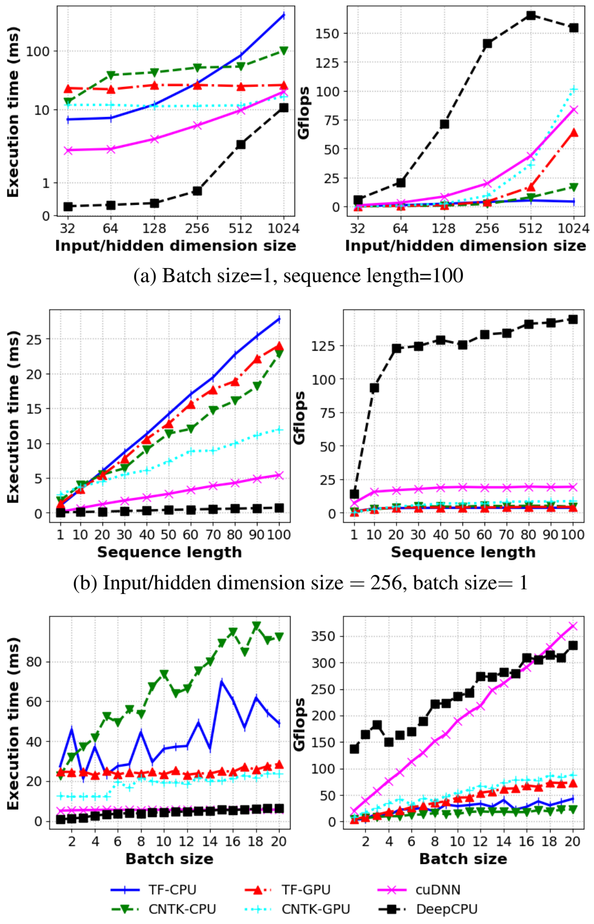
Model parameters				LSTM exec. time (ms)			GRU exec. time (ms)			LSTM speedup		GRU speedup	
input	hidden	batch	seq. len.	TF	CNTK	DeepCPU	TF	CNTK	DeepCPU	TF	CNTK	TF	CNTK
64	64	1	100	7.3	25	0.31	8	25	0.7	26	81	11	36
256	64	1	100	10	27	0.29	9.6	26	0.58	34	93	17	45
1024	64	1	100	19	25	0.42	16	27	0.69	45	60	23	39
64	256	1	100	21	23	0.62	17	30	0.79	34	37	22	38
64	1024	1	100	180	30	6.5	110	37	6.4	28	4.6	17	5.8
1024	1024	1	100	460	33	11	190	40	8.4	42	3	23	4.8
256	256	1	1	0.96	1.1	0.069	0.89	1	0.053	14	16	17	19
256	256	1	10	3.4	2.9	0.16	2.9	3.4	0.14	21	18	21	24
256	256	1	100	28	21	0.74	22	25	0.9	38	28	24	28
64	64	10	100	20	47	1.1	18	43	1.1	18	43	16	39
64	64	20	100	27	74	1.5	25	88	1.5	18	49	17	59
256	256	10	100	51	62	4.4	34	66	3.7	12	14	9.2	18
256	256	20	100	58	91	6.4	51	100	5.4	9.1	14	9.4	19
1024	1024	10	100	400	180	42	280	170	36	9.5	4.3	7.8	4.7
1024	1024	20	100	540	250	68	380	230	60	7.9	3.7	6.3	3.8

Table 1: Execution times and speedups of LSTMs and GRUs, comparing DeepCPU, TensorFlow and CNTK on CPU.

mance gap decreases due to increase in parallelism that allows for an increasing number of GPU cores to kick in. On the other hand, the CPU Gflops plateaus after dimension size of 512.

Varying sequence length. Fig. 5b shows the performance impact of varying input sequence lengths from 1 to 100. As the sequence length increases, the execution time of all implementations except DeepCPU increases almost linearly, or equivalently, their Gflops stays constant. DeepCPU, however, has a sharp jump in performance when sequence length increases from 1 to 10. It demonstrates that DeepCPU exploits data reuse across steps: when sequence length > 1, later steps reuse the weights from the first step, increasing Gflops. Once the sequence length becomes larger than 10, the increase in reuse per flop is marginal. Thus, the Gflops curve is relatively flat when sequence length grows from 20 to 100.

Varying batch size. As shown in Fig. 5c, among all CPU implementations, DeepCPU performs and scales the best with increasing batch size. Among GPU implementations, cuDNN performs significantly better than TensorFlow and CNTK. Comparing DeepCPU with cuDNN, the best CPU versus GPU implementation, DeepCPU is better with small and moderate batch size (< 15) and cuDNN is better with large batch sizes. This crossover is expected. However, as discussed earlier, batch size is often rather small for serving scenarios due to the stringent latency SLA.



(a) Batch size=1, sequence length=100

(b) Input/hidden dimension size = 256, batch size= 1

(c) Input/hidden dimension size = 256, sequence length= 100

Figure 5: LSTM execution time and Gflops with varying input/hidden dimension, sequence length and batch size.

The GPU implementation in existing framework such as TF-GPU has worse performance than cuDNN. This is because TF-GPU and cuDNN do not use the same underlying implementation. In the case of LSTMs, TensorFlow constructs the LSTM operator as a composition of matrix multiplications and activation functions. A single LSTM operator produces hundreds of nodes in the TensorFlow computation graph. While some of these nodes are computed on the GPUs (for example matrix multiplication using cuBLAS), transferring tensors among nodes incurs quite significant overhead. In contrast, cuDNN implementation is a single highly optimized kernel invocation for the entire sequence of the LSTM computation.

6.2. Serving Real-World RNN-Based Models

We evaluate DeepCPU on serving three real-world models. Table 2 provides their RNN specifications.

What's inside DeepCPU? DeepCPU focuses on RNN families and supports LSTM/GRU cell, LSTM/GRU sequence, bidirectional RNN, and stacked RNN networks. It includes fundamental building blocks such as efficient matrix multiplication kernels, activation functions, as well as common deep learning layers such as highway network [57], max-pooling layer [40], multilayer perceptron [50], variety of attention layers [39, 53], and sequence-to-sequence decoding with beam search [59].

Converting trained models into DeepCPU. DeepCPU focuses on serving, and we take a two-step approach to convert trained models (e.g., from TensorFlow/CNTK) to use it. 1) Replace the RNN part(s) of the original model using DeepCPU APIs. In this paper, we implement all three models using DeepCPU C++ APIs. The engineering work is manageable as our library contains many reusable and common components for building neural networks. A more automated way is to integrate our library with an existing framework, which we consider as future work. 2) Port the weights of the trained model to initialize the DeepCPU model instances.

Text similarity (TS). TS measures semantic similarity between texts [45]. It is widely used for grading machine translation results, detecting paraphrase, and ranking query document relevance. It uses bidirectional GRUs to encode text inputs (e.g., sentences) into semantic vectors and measures their relevance with cosine similarity. The GRU computation dominates the performance of the model. The first row in Fig. 6 shows that with DeepCPU, TS runs 12X faster than TensorFlow on CPUs and 15X faster than TensorFlow on GPUs.

Attention sum reader (ASR). ASR extracts single token answer from a given context and can be used for online question and answering [39]. The model uses bidirectional GRU to encode both query and context into semantic vectors and performs reasoning steps to figure out which token in the context is the answer. Fig. 6 shows that DeepCPU reduces ASR serving latency from more than 100ms to less than 10ms, a more than 10X speedup over TensorFlow on CPUs and GPUs.

Bidirectional attention flow model (BiDAF). BiDAF is a high-ranked model on SQuAD reading comprehension competition list [11] for question and answering [53]. It has a hierarchical structure composed of five neural network layers. Among them, three are LSTM-based (Table 3). Fig. 6 shows that DeepCPU reduces the execution time of BiDAF from more than 100ms to less than 5ms, achieving more than 20x speedup against TensorFlow. Table 3 lists the execution time breakdown across layers after the optimization: DeepCPU significantly decreases the execution time of LSTM-based layers.[1]

Correctness. We use TensorFlow for correctness verification: DeepCPU always produces prediction results matching those generated by TensorFlow.

Hardware choice. While not reported in the paper, we have tried DeepCPU on a few different SKUs and processor generations. We have found significant performance improvements even on Haswell and Ivy Bridge generations. The techniques are effective as long as the model is not significantly larger than $L3$ cache of the hardware. DeepCPU also provides additional performance boost from weight-centric streamlining when the weight matrices fit in $L2$ caches of multiple cores.

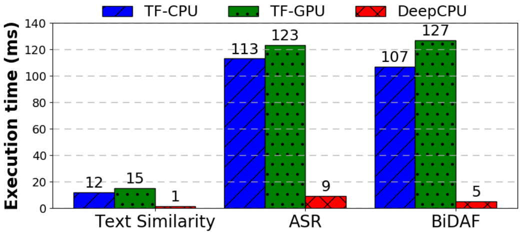


Figure 6: Execution time of TS, ASR, and BiDAF.

Meeting latency SLA with significant cost savings. Besides greatly reducing latency to meet SLA, DeepCPU significantly improves efficiency and reduces serving cost. Take TS model as an example, which is used for ranking query and document pair at our search services. The latency SLA is 6ms and 33 selected documents are ranked for each query. The original TensorFlow model takes 12ms to serve a single ⟨query, document⟩ pair on one CPU machine, violating latency SLA and unable to ship. DeepCPU not only reduces the latency to meet SLA, but more importantly, as shown in Table 4, it only takes 5.6ms to serve a query and *all of its 33 document* on the same machine. *DeepCPU achieves more than 60x throughput gain (i.e.,* $12 \times 33/5.6$*).* Our large-scale search service answers tens of thousands of requests per second, and would originally require more than 10K machines for hosting this model. DeepCPU reduces it to a couple hundred, saving millions of dollars of infrastructure cost just for this model alone.

7. Related Work

DL acceleration library. The closest work to DeepCPU are cuDNN [14, 20] and MKL-DNN [4], which are libraries for accelerating DL frameworks. CuDNN is a GPU library mainly designed for maximizing training throughput, and its performance can be limited by insufficient parallelism when the model size and batch size are small. Other work on optimizing RNNs also target

[1] We also optimized embedding and attention layer to improve end-to-end latency, where the details are beyond the scope of the paper.

Model	RNN parameters
Text similarity [45]	*–input 200 –hidden 512 –source_length 20 –target_length 20 –batch_size 1*
ASR [39]	*–input 200 –hidden 256 –question_length 20 –context_length 100 –context_batch_size 10*
BiDAF [53]	*Phrase embedding: –input 50 –hidden 100 –question_length 15 –context_length 100 –context_batch_size 1*
	Modeling layer (stackd LSTM): –input 800 –hidden 100 –context_length 100 –context_batch_size 1
	Output layer: –input 1400 –hidden 100 –context_length 100 –context_batch_size 1

Table 2: The description of model parameters of RNNs used in real-world models. Sequence lengths refer to maximum sequence length, and both TensorFlow and DeepCPU support variable sequence lengths.

	TF on CPUs	DeepCPU
Embedding + highway	0.69	0.84
Phrase embedding (LSTMs)	13	0.23
Attention layer	13	1.30
Modeling layer (LSTMs)	31	0.90
Output layer (LSTMs)	49	1.50
Total	107	4.77

Table 3: BiDAF execution time (millisecond) per layer.

	T@10	T@15	T@20	T@33
Embedding	0.28	0.24	0.24	0.36
RNN	2.20	2.60	3.40	5.20
Cosine similarity	0.04	0.04	0.04	0.04
Total	2.50	2.90	3.60	5.60

Table 4: Text similarity model execution time where T@K reports execution time of ⟨query, K documents⟩.

GPUs [26, 30, 64]. On CPUs, MKL-DNN is a C/C++ library from Intel to boost DL model performance on Intel architecture, but it only supports convolutional neural networks and has no support for RNNs yet. Other work on multi-core CPUs is similar, targeted more towards CNNs, fully connected neural networks, etc [43, 49, 61]. Some DeepCPU optimizations (e.g., parallelization, fusion) can be generalized to these other networks, whereas optimizations like WCS are more specialized to RNNs.

Compiler and runtime optimizations. There has been work on optimizing DL model performance through compile-time and runtime strategies. Many of them use static analysis to find pipelined operations that can be fused together for improved performance, such as XLA [10], Weld [47], and TensorRT [5]. The compile-time and runtime strategies of these systems are not designed for global optimization of complex structures like RNN sequences. Halide is a domain specific language and compiler to optimize image processing pipeline [48], conveniently separating algorithms with schedules. It is not specially designed for RNN type of recurrent computation, and optimizations such as the weight-centric streamlining cannot be supported easily. It is also hard for its autotuner to search the space efficiently without domain-specific pruning and partitioning methods.

Model deployment. TensorFlow Serving [9] and Clipper [23] are two serving platforms for deploying and serving machine learning models on production systems. Both support caching inference results and batching individual inference requests for better performance. Clipper selects from multiple models to balance latency with accuracy. Our work and these model deployment platforms complement each other: while they focus on the deployment process for serving requests, our library focus on optimizing the inference time of a model itself.

Hardware accelerators. Apart from CPU and GPU, researchers and practitioners are also looking into specialized hardware such as FPGA [42, 46, 54] and ASIC [32, 38], which often require expert hardware designers and long development cycles to obtain high performance. They are not yet widely available commercially.

Model simplification and compression. Existing work shows many model simplification techniques [33, 36, 63] such as sparsifying and quantization that could reduce computation time and space with a small accuracy tradeoff. Co-designing these model optimizations together with system optimizations like those in DeepCPU could present new opportunities to boost performance further.

8. Conclusion

The paper unravels the mystery of poor RNN performance on existing DL frameworks — low data reuse — and develops optimization schemes to reduce latency and improve efficiency of RNN serving. Powered by the new techniques and search strategy, DeepCPU, our serving library on CPUs, improves performance by an order of magnitude, compared with existing work. It transforms many RNN models from non-shippable to shippable with great latency and cost improvement in production.

9. Acknowledgments

We thank great collaborators for their support of DeepCPU, including Junhua Wang, Rangan Majumder, Saurabh Tiwary, Yuesheng Liu, Qifa Ke, Bruce Zhang, Chandu Thekkath, Yantao Li, Jason Li, Fang Liu, and their team. We thank Bin Hu, Elton Zheng, Aniket Chakrabarti, Ke Deng, Doran Chakraborty, Guenther Schmuelling, Stuart Schaefer, Michael Carbin, Olatunji Ruwase, and Sameh Elnikety. The usability and functionality of DeepCPU has been greatly expanded by incorporating their feedback. We thank Kathryn McKinley for reading a previous version of this paper and providing feedback. We thank our shepherd, Deniz Altinbuken, for helping improving the presentation quality of the work.

References

[1] Eigen C++ Library for Linear Algebra. http://eigen.tuxfamily.org.

[2] Intel VTune Amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[3] Intel(R) Math Kernel Library. https://software.intel.com/en-us/mkl.

[4] Intel(R) Math Kernel Library for Deep Neural Networks. https://github.com/01org/mkl-dnn.

[5] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[6] Openblas. www.openblas.net/.

[7] Recurrent Neural Network Tutorial. http://www.wildml.com/2015/09/.

[8] Stream Benchmark. http://www.cs.virginia.edu/stream/ref.html.

[9] TensorFlow Serving. https://www.tensorflow.org/serving/.

[10] The Accelerated Linear Algebra Compiler Framework. https://www.tensorflow.org/performance/xla/.

[11] The Stanford Question Answering Dataset (SQuAD) leaderboard. https://rajpurkar.github.io/SQuAD-explorer/.

[12] Understanding LSTM Networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 265–283, 2016.

[14] Jeremy Appleyard, Tomás Kociský, and Phil Blunsom. Optimizing Performance of Recurrent Neural Networks on GPUs. arXiv preprint arXiv:1604.01946, 2016.

[15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. arXiv preprint arXiv:1409.0473, 2014.

[16] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A Neural Probabilistic Language Model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

[17] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-farley, and Yoshua Bengio. Theano: A CPU and GPU Math Compiler in Python. In *Proceedings of the 9th Python in Science Conference*, pages 3–10, 2010.

[18] Danqi Chen, Jason Bolton, and Christopher D. Manning. A Thorough Examination of the CNN/Daily Mail Reading Comprehension Task. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL '16*, 2016.

[19] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv preprint arXiv:1512.01274, 2015.

[20] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. arXiv preprint arXiv:1410.0759, 2014.

[21] Kyunghyun Cho, Bart van Merriënboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP' 14*, pages 1724–1734, 2014.

[22] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. arXiv preprint arXiv:1412.3555, 2014.

[23] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI '17*, pages 613–627, 2017.

[24] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[25] Bhuwan Dhingra, Hanxiao Liu, Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Gated-Attention Readers for Text Comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL '17*, pages 1832–1846, 2017.

[26] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Y. Hannun, and Sanjeev Satheesh. Persistent RNNs: Stashing Recurrent Weights On-Chip. In *Proceedings of the 33nd International Conference on Machine Learning, ICML '16*, pages 2024–2033, 2016.

[27] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication*, SIGCOMM '13, pages 159–170, 2013.

[28] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. 1990.

[29] Kazushige Goto and Robert A. van de Geijn. Anatomy of High-performance Matrix Multiplication. *ACM Transactions on Mathematical Software*, 34(3):12:1–12:25, May 2008.

[30] Alex Graves. RNNLIB: A Recurrent Neural Network Library for Sequence Learning Problems. https://github.com/szcom/rnnlib.

[31] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech Recognition with Deep Recurrent Neural Networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '13*, pages 6645–6649, 2013.

[32] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 243–254, 2016.

[33] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations, ICLR '16*, 2016.

[34] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep Speech: Scaling up end-to-end speech recognition. arXiv preprint arXiv:1412.5567, 2014.

[35] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[36] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks. In *Advances in Neural Information Processing Systems 29*, pages 4107–4115. 2016.

[37] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, 2014.

[38] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, 2017.

[39] Rudolf Kadlec, Martin Schmid, Ondrej Bajgar, and Jan Kleindienst. Text Understanding with the Attention Sum Reader Network. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL '16*, 2016.

[40] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012*, NIPS '12, pages 1106–1114, 2012.

[41] Nicholas Léonard, Sagar Waghmare, Yang Wang, and Jin-Hwa Kim. rnn : Recurrent Library for Torch. arXiv preprint arXiv:1511.07889, 2015.

[42] Sicheng Li, Chunpeng Wu, Hai Li, Boxun Li, Yu Wang, and Qinru Qiu. FPGA Acceleration of Recurrent Neural Network Based Language Model. In *Proceedings of the 23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '15, pages 111–118, 2015.

[43] Ian Masliah, Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Marc Baboulin, Joël Falcou, and Jack J. Dongarra. High-Performance Matrix-Matrix Multiplications of Very Small Matrices. In *22nd International Conference on Parallel and Distributed Computing, Euro-Par '16*, pages 659–671, 2016.

[44] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent Neural Network Based Language Model. In *11th Annual Conference of the International Speech Communication Association, INTERSPEECH '10*, pages 1045–1048, 2010.

[45] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic Regularities in Continuous Space Word Representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, NAACL-HLT' 13, pages 746–751, 2013.

[46] Eriko Nurvitadhi, Jaewoong Sim, David Sheffield, Asit K. Mishra, Krishnan Srivatsan, and Debbie Marr. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In *26th International Conference on Field Programmable Logic and Applications, FPL '16*, pages 1–4, 2016.

[47] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A common runtime for high performance data analytics. CIDR '17, 2017.

[48] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, 2013.

[49] Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, and Trishul Chilimbi. Optimizing CNNs on Multicores for Scalability, Performance and Goodput. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 267–280, 2017.

[50] Dennis W Ruck, Steven K Rogers, Matthew Kabrisky, Mark E Oxley, and Bruce W Suter. The multilayer perceptron as an approximation to a Bayes optimal discriminant function. *IEEE Transactions on Neural Networks*, 1(4):296–298, 1990.

[51] Hojjat Salehinejad, Julianne Baarbe, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent Advances in Recurrent Neural Networks. arXiv preprint arXiv:1801.01078, 2018.

[52] Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, 2016.

[53] Min Joon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional Attention Flow for Machine Comprehension. arXiv preprint arXiv:1611.01603, 2016.

[54] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From High-Level Deep Neural Models to FPGAs. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '16*, pages 1–12, 2016.

[55] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking State-of-the-Art Deep Learning Software Tools. In *7th International Conference on Cloud Computing and Big Data, CCBD 2016*, pages 99–104, 2016.

[56] Daniele G. Spampinato and Markus Püschel. A Basic Linear Algebra Compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 23:23–23:32, 2014.

[57] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway Networks. arXiv preprint arXiv:1505.00387, 2015.

[58] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014*, pages 3104–3112, 2014.

[59] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014*, NIPS '14, pages 3104–3112, 2014.

[60] AJ Van der Poorten. Continued fraction expansions of values of the exponential function and related fun with continued fractions. *Nieuw Archief voor Wiskunde*, 14:221–230, 1996.

[61] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on CPUs. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS '11*, 2011.

[62] Oriol Vinyals and Quoc V. Le. A Neural Conversational Model. arXiv preprint arXiv:1506.05869, 2015.

[63] Wei Wen, Yuxiong He, Samyam Rajbhandari, Wenhan Wang, Fang Liu, Bin Hu, Yiran Chen, and Hai Li. Learning Intrinsic Sparse Structures within Long Short-term Memory. In *the Sixth International Conference on Learning Representations, ICLR '18*, 2018.

[64] Felix Weninger, Johannes Bergmann, and Björn Schuller. Introducing CURRENNT: The Munich Open-source CUDA Recurrent Neural Network Toolkit. *Journal of Machine Learning Research*, 16(1):547–551, 2015.

[65] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, April 2009.

[66] Geoffrey Zweig, Chengzhu Yu, Jasha Droppo, and Andreas Stolcke. Advances in all-neural speech recognition. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '17*, pages 4805–4809, 2017.

Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs

Yihe Huang[‡] *Matej Pavlovic*[†*] *Virendra J. Marathe*[◇] *Margo Seltzer*[◇‡]
Tim Harris[◇] *Steve Byan*[◇]
Harvard University[‡] *EPFL*[†] *Oracle Labs*[◇]

Abstract

Key-Value (K-V) stores are an integral building block of modern datacenter applications. With byte-addressable persistent memory (PM) technologies, such as Intel/Micron's 3D XPoint, on the horizon, there has been an influx of new high performance K-V stores that leverage PM for performance. However, there remains a significant performance gap between PM optimized K-V stores and DRAM resident ones, largely reflecting the gap between projected PM latency relative to that of DRAM. We address that performance gap with Bullet, a K-V store that leverages both the byte-addressability of PM and the lower latency of DRAM, using a technique called *cross-referencing logs (CRLs)* to keep most PM updates off the critical path. Bullet delivers performance approaching that of DRAM resident K-V stores by maintaining two hash tables, one in the slower (backend) PM and the other in the faster (frontend) DRAM. CRLs are a scalable persistent logging mechanism that keeps the two copies mutually consistent. Bullet also incorporates several critical optimizations, such as dynamic load balancing between frontend and backend threads, support for nonblocking Gets, and opportunistic omission of stale updates in the backend. This combination of implementation techniques delivers performance within 5% of that of DRAM-only key-value stores for realistic (read-heavy) workloads. Our general approach, based on CRLs, is "universal" in that it can be used to turn any volatile K-V store into a persistent one (or vice-versa, provide a fast cache for a persistent K-V store).

1 Introduction

Key-value (K-V) stores with simple Get/Put based interfaces have become an integral part of modern data center infrastructures. The list of successfully deployed K-V stores is long – Cassandra [28], Dynamo [13], LevelDB [30], Memcached [36], Redis [44], Swift [48] – to name just a few. The research community continues to publish K-V store improvements along a variety of dimensions including network stack optimizations, cache management, improved parallelism, hardware extensions, etc. [5, 14, 15, 20, 27, 31, 33, 32, 34, 37, 40, 51, 53, 56]. However, many of these works assume that the K-V store is a volatile cache for a backend database. Most of the persistent K-V stores [7, 13, 18, 28, 30, 34, 51, 44, 48] assume a slow, block-based storage medium, and therefore, marshal updates into blocks written to the file system.

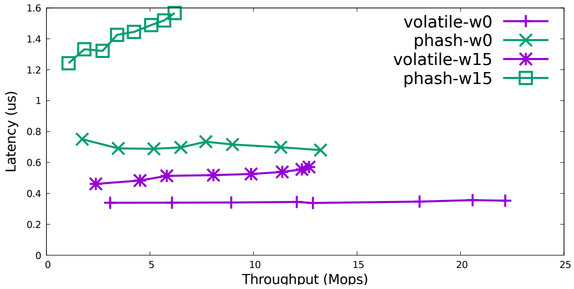


Figure 1: Throughput vs. Latency results of hash table based K-V stores: (i) phash, hosted entirely in emulated PM (Intel's Software Emulation Platform [43, 57]), and (ii) an almost identical K-V store hosted entirely in DRAM (volatile). The emulated PM has 300 nanosecond load latency and bandwidth identical to that of DRAM; DRAM latency is approximately 150 nanoseconds. 0 and 15 represent the percent of K-V accesses that are Puts; keys are selected according to a zipfian distribution. The points on the curves represent the number of threads used in the tests, ranging from 2 to 16 in increments of 2.

At the same time, byte-addressable persistent memory technologies are emerging, e.g., *spin-transfer torque MRAM (STT-MRAM)* [21, 23], *memristors* [46]), and most notably, the Intel/Micron 3D XPoint persistent memory [1]. These technologies provide the persistence of traditional storage media (SSDs, HDDs) with the byte addressability and performance approaching that of DRAM (100-1000x faster than state-of-the-art NAND flash). Byte addressability allows load/store access to persistence (as opposed to the traditional file system interface). As a result, these technologies can profoundly change how we manage persistent data.

The research community has recognized this potential, producing an endless stream of new, PM-optimized K-V stores that leverage PM's byte addressability and low latency, yielding systems that greatly outperform traditional block-based approaches [3, 8, 9, 12, 22, 39, 41, 54, 55, 58]. While this body of work has grown rapidly, most of it ignores the fact that for the forseeable future, PM will be much slower than DRAM [47], making PM resident K-V stores significantly slower than their DRAM counterparts. Figure 1 illustrates the performance gap between K-V stores hosted in DRAM and *emulated* PM. Their implementations differ only in their failure semantics (section 3) and pointer representation (section 6). The 0% writes curves in the graph mirror the $2X$ latency gap between DRAM and emulated PM. This $2X$ gap grows to $3-4.5X$ in the 15%-write case, since writes use expensive persist barriers and transactions for failure atomic updates to the persistent data structures.

Recent PM-based K-V store proposals [41, 54, 55] address this problem by partitioning their data structures between faster DRAM and slower PM, with the DRAM resident structures reconstructed during recovery/warmup. However, these optimizations focus exclusively on B-Tree based indexing structures, not on hash table based structures, which are predominantly used in workloads with `Get`/`Put` point queries. Since these hash tables are central to many popular K-V stores [30, 36, 44], leveraging both DRAM and PM in their implementations is critical to their performance.

We present Bullet, a new K-V store designed for multi-/many-core systems equipped with persistent memory. Bullet explicitly leverages the combination of fast DRAM and slower, byte-addressable PM, to deliver performance comparable to that of a DRAM resident K-V store in realistic workloads. Bullet's architecture is designed to handle most, if not all, client requests in the faster DRAM, minimizing the number of PM accesses on the critical path. This naturally leads to an architecture with a DRAM resident cache, similar to the approach taken by traditional databases and K-V stores. However, Bullet deviates from traditional approaches in that the cached *frontend* hash table and the persistent *backend* hash table representations are virtually identical – differing only in their pointer representations (section 6) and failure handling semantics. This facilitates efficient access to backend data whenever there is a miss in the frontend – PM's byte addressability plays a critical role in making this possible.

We keep the frontend and backend mutually consistent by employing a novel, efficient, and highly concurrent logging scheme, called *cross-referencing logs (CRLs)*. In an architecture using per-thread persistent logs, CRLs track ordering dependencies between log records using simple cross-log links instead of synchronizing the threads' log access [29, 52]. Bullet processes `Get` requests exclusively in the frontend, without log access. On their critical path, `Put` requests access the frontend as well, while also writing log records to CRLs. This results in a single thread-local log append per update.

Backend threads, called *log gleaners*, apply persisted log records to the backend hash table. We use an *epoch* based scheme to apply log records to the backend in batches. The epoch based scheme's primary purpose is to enable correct log space reclamation. The backend's hash table updates must be applied in a crash consistent manner. We address this problem using a backend runtime system [35] that supports *failure atomic transactions* similar to several other persistent memory transaction runtimes [16, 50]. The resulting code path is complex, but *not on the critical path* of client requests.

We apply four key optimizations in Bullet: 1) fully decoupling frontend execution from PM performance on `Put` operations, 2) nonblocking `Gets`, 3) dynamic thread switching between the frontend and backend, based on the write-load in the system, and 4) opportunistic `Put` collapsing. Our base design, coupled with these optimizations, make Bullet's performance close to that of a DRAM resident K-V store: For realistic, read-heavy workloads, Bullet either matches or comes close to the performance of a DRAM-resident volatile K-V store, delivering throughput and latency $2X$ better than that of a state-of-the-art hash table based K-V store, HiKV [54], on a system with emulated PM whose access latency is $2X$ of DRAM access latency. For pathological write-heavy workloads, Bullet's throughput is comparable to or better than that of HiKV and its operations' latency is approximately $25-50\%$ lower. Relative to a volatile K-V store, Bullet's latency and throughput degrade by approximately 50% under write-heavy workloads.

2 Bullet's Architecture

2.1 Overview

Figure 2 depicts the high level architecture of Bullet, separated into the frontend and backend components, each of which contains almost identical hash tables. The frontend resides in the *volatile domain* (DRAM). It contains a configurable number of threads that process incoming requests, applying them to its hash table. Each frontend thread additionally "passes on" update requests to the backend, by appending update requests to a *thread-local* persistent log. An update completes when it has been safely written to the log. The backend resides in the *persistent domain* (PM). The backend's *log gleaner* threads periodically read requests from their corresponding persistent logs and apply them to the persistent hash table, in a failure-atomic and correctly ordered manner. In this "base" configuration, each persistent log maps both to a *log writer* thread in the frontend and a log gleaner thread in the backend.

While processing client requests, a frontend thread first looks up the target key in the frontend K-V store. If the lookup succeeds, the frontend applies the operation. If it is an update (`Put` or `Remove`), the thread also appends the `<opcode,payload>` tuple to its persistent log. If the lookup in the frontend fails, the thread issues a lookup to the backend. A successful lookup creates a copy of the key-value pair in the frontend, at which point the operation proceeds as if the original frontend lookup succeeded. If the lookup fails: (i) a `Get` returns a failure

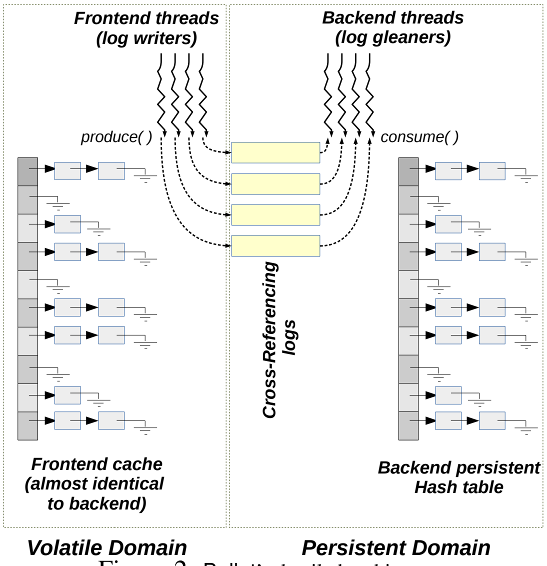


Figure 2: Bullet's detailed architecture.

code to the client, (ii) a `Put` inserts the pair into the frontend, including the log write, and (iii) a `Remove` returns with a failure code.

The rest of this section details our solutions to a number of technical challenges: persistent and volatile hash table implementation (subsection 2.2), the parallel logging scheme (subsection 2.3), correct coordination between frontend and backend threads (subsection 2.4), and failure atomic updates (section 3).

2.2 Hash Tables

As shown in Figure 2, Bullet's frontend hash table is in DRAM and therefore volatile. It supports the standard K-V operations: `Get`, `Put`, and `Remove`. The hash table is similar in structure to other key-value stores [36]: It is closed addressed, with chaining to handle hash conflicts. It grows via a background thread responsible for doubling the number of hash table buckets when occupancy crosses a threshold size (twice the number of buckets). Regular operations can occur concurrently with growing the table. Each hash table bucket has its own reader-writer spinlock for thread synchronization – lookups acquire the spinlocks for reading (shared), and updates acquire the spinlocks for writing (exclusive).

The backend hash table is structurally identical to the frontend one, with its own per-bucket chains and spinlocks. However, unlike the frontend (volatile) hash table, the backend hash table resides in persistent memory and must survive failures. Bullet uses failure atomic transactions for `Put` and `Remove` operations to provide this guarantee (section 3). `Gets` execute identically to those in the frontend (except a failure to find a key is always a failure in the backend, while the frontend has to check the backend before failing).

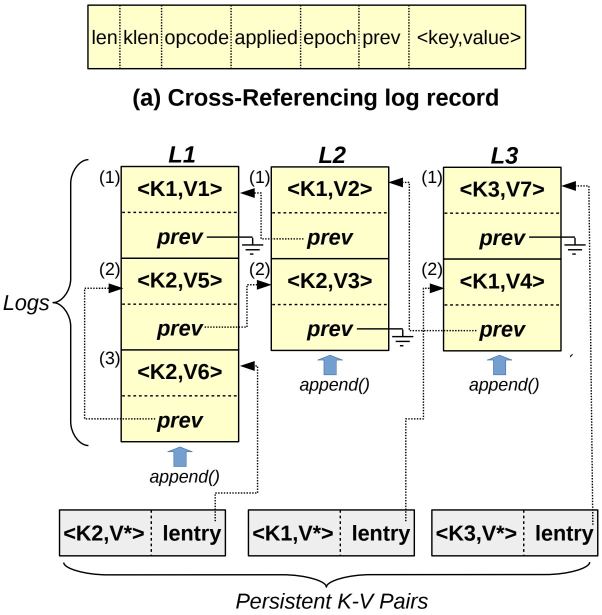
The per-bucket spinlocks in the persistent hash table are used only for synchronization between concurrent backend threads and are semantically volatile. We found placing the spinlocks in the bucket extremely convenient, with the added benefit of improved cache locality compared to an alternative where the spinlocks are mapped elsewhere in DRAM. Since a bucket's spinlock resides in persistent memory, its state can persist at arbitrary times (e.g., due to cache line evictions). A failure could leave a spinlock in the *locked* state. We leverage a generation number technique [10] to reinitialize such locks after a restart – Bullet increments a global persistent generation number during every warm-up and compares that generation number to a generation number contained in every lock. If the generation numbers do not match, Bullet treats the lock as available and reinitializes it.

2.3 Cross-Referencing Logs

The frontend communicates updates to the backend via a log. In a conventional, centralized log design [25, 26, 38], the log becomes a bottleneck, because concurrent updates must all append records to the log. Thread-local logs neatly address this contention problem, but introduce a new challenge: records from a multitude of logs must be applied to the backend in the correct order – the order in which the corresponding operations were applied in the frontend. While prior systems partition the key space so that all updates to a particular K-V pair appear in the same log file (e.g., [34]), Bullet does not partition the data and K-V pair updates can happen in any thread. This way Bullet is not susceptible to load balancing issues encountered in partitioned K-V stores [34]. We address the ordering problem in a different way: We introduce *cross-referencing logs (CRLs)*, to provide highly scalable, concurrent logging on PM, without relying on centralized clocks [29, 52] to enforce a total order of update operations.

Figure 3 illustrates CRLs. Each frontend log writer thread maintains its own persistent log. Logically, each log record is a `<opcode,payload>` tuple. Opcode allows the application to define high-level operations expressed by each log record. For example, when Bullet manages a hash table of lists, each list append can be expressed by a single log record, where the opcode refers to the list append operation, and the payload contains the record identifier (a reference to the list in question) plus the value to be appended. The order in which non-commutative operations like this are applied is important, hence the necessity of the CRL scheme. The logs require no synchronization on appends, because there is only one writer per log. The backend maintains corresponding log gleaner threads that consume log records and apply them to the backend persistent hash table in a failure-atomic manner.

The logs are structured so that log gleaners can easily



(a) Cross-Referencing log record

(b) Cross-Referencing log structure

Figure 3: Cross-Referencing Log (CRL) architecture.

determine the correct order in which to apply log records. Figure 3a shows the log record layout. The `len, klen, opcode, <key,value>` fields contain the information implied by their names. The `applied` field contains a flag indicating whether the backend K-V store accurately reflects the log record. The `prev` field contains a persistent pointer to the prior log record, if one exists, for the given key. We defer discussion of `epoch` until subsection 2.4.

Appending Log Records: Figure 3b depicts three logs $L1$, $L2$, and $L3$ containing log records for keys $K1$, $K2$, and $K3$. The `lentry` field of the persistent key-value pairs (shown at the bottom) contains a persistent pointer to the most recent log record for the key-value pair. Thus, the list formed by the `lentry` and `prev` pointers represents the evolution of a key-value pair in reverse chronological order, where the log record containing a NULL `prev` pointer indicates the first update to the pair present in any of the logs. The list for a specific key can *criss-cross* among multiple logs, hence the name *cross-referencing logs*. For instance, log records for key $K1$ appear in all three logs, whereas log records for key $K2$ appear only in logs $L1$ and $L2$.

Before a log writer appends a log record, it acquires the key's hash bucket lock, to ensure that it is the only writer for the target key-value pair. Then, the writer (i) populates the log record at the tail end of the log, setting the log record's `prev` field to the value stored in the K-V pair's `lentry`, (ii) persists the log record, (iii) updates and persists the log's tail index, and finally (iv) updates and persists the key-value pair's `lentry` pointer, thus completing the linked list insertion. In all, an append requires 3 persist barriers.

Applying Log Records: Gleaner threads periodically scan logs and apply log records to the backend hash table in a failure-atomic manner. A log gleaner starts processing from the beginning of the log (the head). For each log record encountered, the gleaner looks up the corresponding key-value pair in the backend persistent hash table; a new key-value pair is created if necessary.

The gleaner retrieves the key-value pair's `lentry` to process all existing log records for that key-value pair. At this point, we need to ensure that at most one gleaner is processing log records for a given key-value pair. To that end, we add another spinlock that enables only one gleaner to apply all the log records for a key-value pair. This spinlock is placed in the key-value pair itself. A gleaner must acquire this spinlock before processing the log records for the key-value pair. The gleaner then traverses to the end of the list, checking the `applied` flag of each log record to determine the point from which the gleaner needs to apply log records. Upon finding the last (chronologically the first) unapplied log record, the gleaner applies the log records in the chronological order determined by the linked list (i.e., in the reverse order of the list). The gleaner sets the `applied` field after applying the log record to the persistent hash table. We discuss the transaction mechanism that ensures recoverability of these updates in section 3.

After applying all the log records for a key-value pair, the gleaner can reset `lentry` to `NULL`. This however races with a frontend log writer's append for the same key-value pair, which requires an update to the key-value pair's `lentry`. Fortunately the data race can be avoided using a `compare-and-swap` instruction, by both the appender and the gleaner, to atomically change `lentry`.

Consider the example in Figure 3. A gleaner for log $L1$ will first encounter log record labeled (1). It uses the log record's key, $K1$, to retrieve the corresponding persistent key-value pair (at the bottom of Figure 3b). From that key-value pair object, the gleaner begins at the end of the log record list at $L3(2)$, then continues to $L2(1)$ and finally $L1(1)$. It then applies each of those log records in reverse traversal order.

Handling Removes: Removes are unique, in that they logically require removing a record at the front end, but the same record at the persistent back end may not be removed at the same time due to log delays. If a deletion is followed by a re-insertion of the same key, the front end and back end can grow inconsistent, due to the fact that CRL relies on the back end record to generate cross-references. To address this problem, we keep the front end record alive as long as we need to by using a special "tombstone" marker. Appending a delete log record only sets the tombstone marker at the front end, but does not remove the record. Future look-ups on the front end regarding this record now return “not found”, until a reinsertion clears this tombstone marker. The front end records marked with tombstones are only physically removed when the corresponding records in the back end are removed during the log gleaning phase.

Rationale: While cross-referencing logs are interesting, one could argue that the criss-crossing could lead to bad cache locality for log gleaner threads. However, it is a trade-off – a thread may suffer poorer locality in its log traversal, but it enjoys superior cache locality, by repeatedly acting upon the same key-value pair. This cache benefit is further enhanced, because log records are concise representations of operations, but the operations themselves tend to lead to “write amplification”, accessing and updating many more memory locations than a single log entry. By continuing to operate on the same key-value pair, we observe that those accesses are far more likely to produce cache hits. Additionally, gleaners never block behind other gleaners. If a gleaner detects that the key-value pair it needs to process is already locked by another gleaner, it can safely assume that the spinlock owner will apply the log record. As a result, the gleaner simply skips that log record. This approach works for the *fail-stop* failure model we assume – a failure terminates the entire key-value store process.

2.4 Log Space Reclamation

The cross-referencing logs that act as bridges between Bullet’s frontend and backend do not grow indefinitely. In fact, they are circular logs and contain persistent `head` and `tail` indexes. To keep the system running without interrupt, Bullet must recycle log space.

The log gleaners work in phases or *epochs*. Between epochs, the gleaners wait for a signal from the *epoch advancer* thread, which periodically tells the gleaners to start applying logs records. Each gleaner reads the log, beginning at the head, and applies the log records as described above. However, it does not advance its log’s head index. Instead, the epoch advancer periodically terminates the current epoch by telling the gleaners to stop processing the log. At this point, the epoch advancer updates each gleaners’ head index. If a log writer fills the log more quickly than the corresponding log gleaner applies the log, the log can fill. If this happens, the writer blocks until the gleaner frees space in the log.

3 Failure Atomic Transactions

To ensure a consistent state after system failure, the backend’s hash table updates must be failure atomic. We use failure atomic persistent memory transactions. Similar to prior work [6, 10, 16, 50], we developed a persistent memory *access library* [35], which contains support for low level programming abstractions that greatly simplify application development for persistent memory. Our access library supports transactions that provide failure atomicity guarantees for updates to persistent memory.

```
pm_txn_t *txn_begin();
txn_state_t txn_commit(txn);
void txn_read(txn, src, len, dst);
void txn_write(txn, dst, len, src);
... // other accessor functions
pm_region_t *pm_region_open(path);
void pm_region_close(region;
void *pm_get_region_root(region);
void pm_set_region_root(region,addr);
... // other region management functions
void *pm_alloc(txn, len);
void pm_free(txn, addr);
```

Figure 4: Base persistent transactions API.

Figure 4 presents our transaction runtime’s API. The interface provides `txn_begin` and `txn_commit` functions to delineate transaction boundaries and various `txn_read` and `txn_write` accessor functions for transactional reads and writes of persistent data. The interface also provides transactional variants of general purpose `libc` functionality, such as `memcpy`, `memset`, `memcmp`, etc. We provide “flat nesting” semantics [19]. The transaction mechanism provides *only* failure atomicity semantics; it does not transparently manage concurrency control, as do some software transactional memory runtimes [10, 50]. Bullet itself performs the necessary synchronization to avoid data races and deadlocks.

The access library also provides a *persistent region* abstraction [6, 10, 50]. The persistent region builds over the `mmap` interface, mapping a persistent memory file into the application’s address space [49]. The persistent region contains a persistent heap, modeled after the Hoard allocator [4, 50]. Application data hosted in a persistent region can be made reachable via a special, per region, *root* pointer. Bullet uses the region’s root pointer to reach its persistent hash table and cross-referencing logs. Finally, the access library uses redo logging [16, 35, 50] to implement failure atomic writes.

4 Optimizations

4.1 Tightening the Update Critical Path

Bullet is designed to streamline critical paths of update operations. To that end, Bullet moves the persistent hash table’s failure-atomic updates off the critical path. However, the design presented thus far does not entirely remove transactions from the update critical path. On a `Put` operation, if the key does not exist in either the frontend or backend hash tables, Bullet allocates a new *persistent* K-V pair object, storing a reference to it in the log record. Furthermore, when the persistent log append completes, we must also update the the key-value pair’s `lentry` to reference that newly created log record. Accessing the persistent K-V pair itself requires a lookup in the backend hash table, which is costly due to the relatively slower persistent memory. All these accesses and updates contribute significant latency to the frontend update operations.

We address this problem by completely decoupling backend data accesses from the frontend update operations, by moving the `lentry` pointer to the frontend hash table's K-V pair. This gets rid of the requirement to locate, and possibly allocate, the backend's K-V pair for a new key. It also eliminates the expensive persist barrier required to persist the `lentry`, since it is no longer persistent; it's part of the volatile copy of the K-V pair. This also eliminates the need for transactions in the frontend, thereby considerably shortening the frontend's update critical path.

4.2 Nonblocking `Gets`

Bullet's "base" version, as described in section 2, uses reader-writer locks to synchronize access to the frontend and backend buckets. While these work well with few frontend and backend threads, they do lead to increased cache contention between concurrent readers on the lock's readers counter – the lock implementation uses a signed integer, where a value greater than 0 indicates one or more readers, and a -1 indicates a writer. The resulting cache contention can restrict scalability. This can be especially pronounced in workloads where accesses follow a power-law distribution and are skewed to a small set of K-V pairs, as is experienced by real world K-V stores [15, 40].

As in prior work [15], we support nonblocking `Get` operations. The principal hurdle for nonblocking `Gets` is memory reclamation – a `Put` or `Remove` can deallocate an object being read by a concurrent `Get`. We need support to *lazily* reclaim the removed objects. Bullet's epochs neatly enable this lazy memory reclamation. The epoch advancer thread periodically increments Bullet's global epoch number. Each frontend thread maintains a local epoch equal to the global epoch number at the beginning of an operation.

When freeing an object, the frontend thread enqueues the object on its local free queue. The enqueued node contains a pointer to the object and the thread's epoch number. On each enqueue, the frontend thread frees the head node of the queue if its epoch is older than the smallest epoch of all the frontend workers. The smallest epoch is a conservative approximation of workers' epochs – it is computed periodically by the epoch advancer thread at the end of each epoch.

Additionally, we structure the frontend hash table's overflow list similar to prior nonblocking concurrent lists [17] so that a reader does not get stuck in a cycle if the node it is accessing is removed from the list by a concurrent writer. While reads are nonblocking, concurrent writers do synchronize with each other on the bucket's spinlock.

4.3 Managing Writer and Gleaner Counts

In the base design, Bullet contains a static mapping between frontend writers, logs, and backend gleaners. Although this approach avoids synchronization among writers and gleaners, it wastes CPU cycles if there is a mismatch in the rates of log record production and consumption. We need to decouple these three parts of Bullet to let threads dynamically perform the roles of frontend and backend based on the write load.

4.3.1 Decoupling Writers from Gleaners

Maximizing Bullet's throughput requires that we keep all threads busy. In practice, this requires that we relax the 1:1 mapping between writers and gleaners. We permit each writer/gleaner to append/consume entries to/from any log. This way we achieve optimal throughput by setting the writer/gleaner ratio according to the ratio of the respective rates of production/consumption of log entries.

Although this requires synchronization among both writers and gleaners, we make the overhead negligible, by coarsening switching intervals between writers and gleaners. Writers lock their log and keep the lock as long as the log is not full. When a log fills, the writer unlocks it and switches to the next free log not currently in use. The same thing happens for gleaners; they switch logs when they have no work to do. For log sizes on the order of megabytes, these switching events are rare enough not to impact performance in an observable way.

4.3.2 Dynamic Adjustment of Writer/Gleaner Ratio

One drawback of the preceding approach is that, selecting the correct writer and gleaner counts, requires knowing the rates of producing and consuming log entries. However, these rates depend heavily on the workload (read/write ratio, key distribution), and the relative performance of DRAM and persistent memory. For example, a write-heavy workload on a machine with a slow persistent memory generally requires more gleaners than a read-heavy workload.

To achieve high throughput in as many scenarios as possible, threads dynamically change their roles, writing or gleaning depending on what is currently needed. The advantage of this approach is twofold. First, it makes Bullet suitable for a wide range of workloads, without prior profiling and configuration. Second, the system adapts to dynamically changing workload, maintaining near optimal throughput throughout.

The key for achieving optimal throughput is preventing the logs from becoming full (writers stalling) or empty (gleaners stalling). To this end, we periodically check (once per epoch) the occupancy of the logs. If the log occupancy passes a pre-defined threshold of 60%, we switch one thread from writing to gleaning. If, upon the next check, the occupancy is still increasing, we add yet another gleaner. We repeat this until the log occupancy starts decreasing. The inverse happens when the log occupancy drops below 30%, in which case we start moving gleaners back to writing.

Making threads switch between worker and gleaner roles is an interesting control theory problem by itself. Our algorithm evolved over several attempts at simpler approaches, which failed to achieve both stability (i.e., avoid frequent role switching) and responsiveness.

4.4 Collapsing `Put` Operations

Recall that multiple updates to the same key result in a linked list of log records. Gleaners traverse the chain and apply all the log records from oldest to newest (see subsection 2.3).

However, it is not necessary to apply every `Put` operation, since the most recent `Put` overwrites the effects of all older `Puts` and `Removes`; same is the case with `Removes`. Thus, a gleaner applies only the newest operation in a chain of log records, without following back pointers at all. To prevent a newer value being overwritten by an older one, a gleaner applies a log record only if it contains the globally newest update for the corresponding key. To determine whether a log entry is the newest for its key, the gleaner checks the corresponding K-V pair's `lentry` pointer, as this always points to the key's newest log record.

Collapsing updates appears to make the criss-cross log record links unnecessary. However, this is the case only for *idempotent* updates, e.g. `Put` and `Remove`. We however plan to extend Bullet to support non-idempotent updates similar to recent data structure stores like Redis [44], where the criss-cross links will be required for correctness.

5 Recovery and Warmup

Recovery is simple for Bullet. Since updates must complete in the frontend before we apply them to the backend and the frontend disappears on failure, Bullet never has anything to undo. In theory, recovery entails two parts: 1) reinitializing the frontend DRAM resident state and 2) applying log records in the CRLs to the backend. Bullet's architecture however permits us to eliminate all of step 2 from recovery, and reduce step 1 drastically: During recovery, the CRLs' log records can be applied to the frontend hash table, instead of applying them to the backend. This has the nice side effect that there is no special recovery code for the backend. We assume that recovery for the backend's persistent transactions happens before Bullet's recovery is triggered. Application of CRLs to the backend is relegated to the normal gleaning process.

Note that recovery itself "warms up" the frontend hash table with key-value pairs found in the CRLs. Thereafter, misses in the frontend populate the corresponding key-value pairs from the backend as described in subsection 2.3. Thus warmup time and recovery time are one and the same and are proportional to the time taken to apply the CRLs.

6 Implementation Notes

We implemented Bullet in C++ and used our PM access library (section 3) developed in C. We used `pthreads` to implement both the frontend and backend threads. The frontend K-V store uses the `jemalloc` library to handle memory allocations. For the backend, we rely on the access library's heap manager, which is based on the scalable Hoard allocator [4, 50].

The PM access library presents to Bullet a persistent memory hosted `mmap()`ped file as a persistent region. Bullet's persistent domain is precisely that region. The `mmap` dependency means that the address of the persistent domain is unpredictable. Therefore, we must represent persistent pointers in a manner amenable to relocation, so we represent persistent pointers as offsets from the region's base address.

Bullet's backend contains a *root structure* that hosts persistent pointers to the persistent hash table and the cross-referencing logs. Wherever we do not use persistent transactions, we carefully order stores and persists to persistent data structures (e.g. CRL appends, initializing a newly allocated key-value pair) for crash consistency.

All update operations in Bullet's backend threads use transactions to apply CRL log records to the backend hash table. In contrast, Bullet's frontend updates need not be transactional; they need only append records to the the CRLs. This indicates two different implementations for all update operations (e.g., frontend and backend implementations of `Put`, `Remove`, etc. operations). This doubles the coding effort for these operations.

The access library's transactional runtime uses Intel's persistence enforcement instructions [24] – cache-line writeback (`clwb`) and persist barrier (`sfence`) instructions to correctly order transactional writes to PM. CRL appends also use these instructions: first, we write back the cache lines of the updated log record using `clwb` and then persist them using `sfence`. Next, we update and persist the log's tail index using the same instructions.

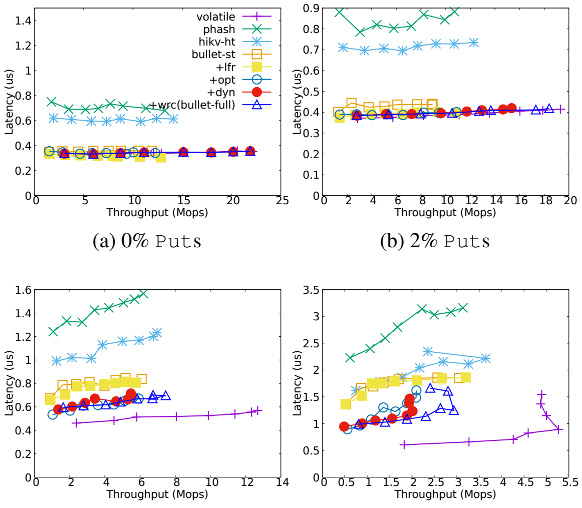
7 Evaluation

We evaluated Bullet's performance on Intel's Software Emulation Platform [43, 57]. This emulator hosts a dual socket 16-core processor, with 512GB of DRAM, of which 384GB is configured as "persistent memory". Persistent memory is accessible to applications via `mmap`ped files hosted in the emulator's PMFS instance [43].

The aforementioned persistence instructions, `clwb` and `sfence`, are not supported by the emulator. We simulated `clwb` with a `nop` and the `sfence` with an idle spin loop of 100 nanoseconds. We expect these to be reasonable approximations since `clwb` is an asynchronous cache line writeback, and an `sfence` ensures that prior writebacks make it to the memory controller buffers, which we assume to be a part of the memory hierarchy's "persistence domain" [45] – 100 nanoseconds is the approximate latency to the memory controller buffers on the emulator. The emulator does support configurable `load` latency to persistent memory; we set it to 300 nanoseconds, twice the load latency of the DRAM on the machine [57]. We configured the PM to have the same bandwidth as that of the emulator's DRAM. We experimented with a lower bandwidth option (1/4 of DRAM bandwidth, which was the only other available option on the emulator), but obtained identical results, suggesting that our experiments did not saturate the memory bandwidth available on the emulator (36 GB/s).

We conducted an 8-way evaluation to see how effectively Bullet eliminates the gap between DRAM and PM performance. The eight systems were as follows. 1) A DRAM-only version that uses just the frontend hash table (volatile), which places an upper bound on performance. 2) A PM-only version that uses Bullet's backend hash table (phash), providing a lower bound on performance. 3) hikv-ht, our implementation of the hash table component of HiKV – a state-of-the-art K-V store, whose hash table resides in PM [54]. HiKV gets a somewhat unfair advantage in our experiments, because it does not ensure that the state of the persistent memory allocator persists. However, the allocator's state can be rebuilt after a restart from HiKV's hash table, although we have not implemented this. 4) bullet-st, the base version of Bullet, which assigns frontend and backend threads statically and uses transactions in the critical path of update operations. 5) +lfr, the base version of Bullet with optimized, lock-free `Get`s. 6) +opt, the version of Bullet that additionally eliminates failure atomic transactions from the critical path of update requests. 7) +dyn, the Bullet version that, along with above optimizations, supports dynamic thread switching between the frontend and backend. 8) bullet-full (also appears as +wrc(bullet-full) in the graphs), the full Bullet version that additionally contains the write collapsing optimization. Although the frontend of Bullet can be a subset of the backend, in our experiments the frontend is a full copy of the backend.

We evaluate various aspects of Bullet comprising scalability and latency, dynamic behavior of worker threads, and log size sensitivity in a microbenchmark setting. In all our experiments, `Get`/`Put` requests are drawn from a pre-created stream of inputs with a zipfian distribution of skewness 0.99, which is the same as YCSB's input distribution [11]. We average over five test runs for each data point. We also use an evaluation framework that uses independent clients to better understand end-to-end performance of these systems as client load increases. The clients are independent threads residing in the same address space as Bullet and communicate requests and responses through globally shared request/response buffers. We do not use clients communicating with Bullet over TCP connections, since the network stack latency itself tends to significantly mute important performance trade offs between the evaluated K-V stores [14, 54].



(a) 0% `Puts`

(b) 2% `Puts`

(c) 15% `Puts`

(d) 50% `Puts`

Figure 5: Latency (99th percentile) vs. Throughput results. Each point on the curves represents a different number of worker threads ranging from 2 to 16 in increments of 2.

7.1 Latency vs. Throughput

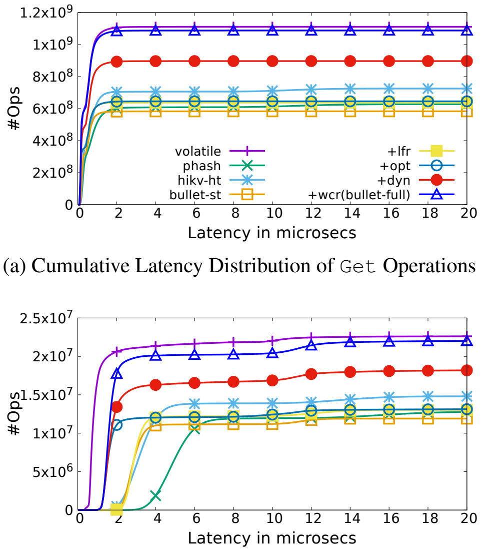
Figure 5 shows performance as a latency/throughput tradeoff under workloads whose write percentage varies from 0% (read-only) to 50% (write-heavy). We begin by creating a 50-million key/value pair store with 16-byte keys and 100-byte data values; these choices are in line with what is observed in real-world settings [2, 54]. Each experiment runs a specified number of of worker threads with the requested read/write ratio, using `Get`/`Put` operations (`Remove` performance is comparable to that of `Put`). Each worker selects key-value pairs from the prepopulated zipfian stream of keys and performs the selected operation. The worker continuously repeats these operations for 1 minute (we experimented with 5 – 10 minute runs, but the results were unchanged).

For the dynamic worker role versions of Bullet (+dyn and bullet-full), some workers switch roles to become backend log gleaners. In such cases, the worker posts its current unapplied operation on a globally visible queue of requests, so that some other frontend worker will process it (to ensure forward progress, we guarantee that at least 1 worker remains in the frontend). We measure latency of only those operations that have a frontend worker assigned to them (the requests posted in the central queue are a rare occurrence and are processed relatively immediately by frontend worker threads).

Notice the clear impact of slower PM on the 0% `Put` case in Figure 5a. The difference between phash's and volatile's latency and throughput mirrors the difference in PM and DRAM latency. hikv-ht performs noticeably better than phash, owing to some of its cache locality oriented optimizations. But these marginal improvements suggest that additional optimizations cannot eliminate the fundamental problem of slower PM. All of Bullet's versions' latencies align almost exactly with volatile. bullet-st shows slight overhead associated with lock-based Gets. All static worker role assignment variants of Bullet (bullet-st, +lfr, +opt) effectively end up using just half of the available workers in the frontend and produce throughput approximating half the throughput of volatile; the backend worker threads effectively waste CPU cycles. Our dynamic worker assignment framework (in +dyn, +wrc(bullet-full)) correctly assigns all workers to the frontend, which performs comparably to volatile.

The 2% Put test is more representative of real-world (read-dominated) workloads [40]. As Figure 5b shows, the relative latency differences remain similar; there is a small increase in the absolute latencies reflecting effects of longer latency Put operations. For the same reason, the absolute throughput numbers are smaller, but the relative difference between volatile, phash, hikv-ht, and the static variants of Bullet remains the same. However, in +dyn and bullet-full we begin to see the impact of logging. The primary source of these overheads is the dynamic switching of 1 or 2 worker threads between the frontend and backend. Note that even with 2% Puts, our CRLs quickly cross the occupancy threshold of 60%, which forces frontend threads to incrementally switch to the backend log gleaner roles if the occupancy keeps growing across epochs. A consistent rate of 2% Put traffic is large enough to force at least one worker to stay a log gleaner through the entire execution. +dyn's performance drops by a significant 25% compared to volatile. However, our write collapsing optimization works exceptionally well to significantly reduce that margin to about 5%: the zipfian distribution of requests allows for substantial write collapsing (30 – 50%), which leads to the log gleaner applying the log more quickly, spending the saved time in frontend request processing.

The 15% workload, shown in Figure 5c, illustrates more clearly the impact of the different optimizations. Compared to volatile, Bullet's bullet-st and +lfr versions show a 40% degradation in latency. The failure atomic transactions used for Put operations of these versions are primarily responsible for this degradation. This degradation is mitigated by half with our critical path optimization present in Bullet's +opt, +dyn, and bullet-full versions. Latency of the PM-only K-V stores, phash and hikv-ht, is approximately 3X and 2.5X higher than that of volatile. Notice the throughput of Bullet's dynamic versions drops significantly. With 15% Puts, we observed a larger fraction (4 – 6) of worker threads getting forced to operate as log gleaners in the backend for the entire duration of the test. That leads to a significant reduction in overall throughput, since threads migrated from the frontend to the backend do not process new requests.



(a) Cumulative Latency Distribution of Get Operations

(b) Cumulative Latency Distribution of Put Operations

Figure 6: Get, Put Cumulative Latency distributions on 16-thread test runs with 2% Puts.X

With the even higher 50% Put rate of Figure 5d, we observe additional interesting behavior. The variants that use transactions in their Put critical paths exhibit significantly increased latency, approaching that of hikv-ht's latency. The rest of Bullet's variants (+opt, +dyn, and bullet-full) exhibit lower latency, which starts to grow only as the set of worker threads grows. We attribute this performance degradation to cache contention between frontend and backend threads. Notice that the working sets of the frontend and backend threads are largely different – a frontend log writer accesses the frontend hash table and a log, whereas the colocated (on the same socket) backend log gleaner accesses the backend hash table and possibly a different log. The more threads there are, the greater the cache contention, and the worse the performance. Overall, the results suggest that workloads with very high write rates are not a good fit for Bullet.

7.2 Latency Distribution of Gets and Puts

Figure 6's segregated cumulative latency distribution graphs for Gets and Puts provide deeper insight into the behaviour of the K-V stores. Figure 6a shows latency of Gets. The phash and hikv-ht latencies average to about 450 and 380 nanoseconds respectively, whereas volatile and all of Bullet's versions average to 220 nanoseconds. Average latencies of Puts are more scattered: volatile is the fastest with 750 nanoseconds, followed by Bullet's versions that do not contain trans-

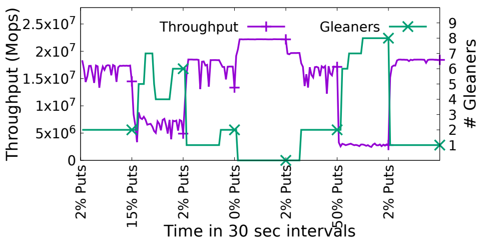


Figure 7: Variation in throughput and log gleaner count as the write load on bullet-full changes. The load, shown on the X-axis labels, switches every 30 seconds. The left Y-axis shows throughput for every second, and the right Y-axis shows the number of log gleaners at the end of each second.

actions in the critical path (at 1 microsecond), followed by Bullet's versions that contain transactions in the critical path (at 2.5 miroseconds). HiKV's latency matches that of Bullet's versions with transactions on the critical path. phash is the slowest with latency averaging to 5 microseconds; this is an $8X$ slowdown compared to volatile. Note that the backend `Put` operations in all of Bullet's versions apply the same `Put` operation used in phash. This largely explains the significantly higher cost associated with applying log records to the backend, and why as little as 2% `Puts` can force worker threads to play the log gleaner role for much longer durations that amplify to a minimum of 10% slowdown in throughput compared to volatile in Figure 5b.

7.3 Dynamic Behavior of Workers

Figure 7 shows bullet-full's dynamic worker role framework in action. It reports the throughput as well as the gleaner count at the end of every second, over a duration of 210 seconds. Every 30 seconds, we change the load of `Puts` on bullet-full. After a warmup phase of 30 seconds of 2% `Put` rate, we vary the `Put` rate between 2-15-2-0-2-50-2%, in that order. As is clear from the graph, our dynamic worker role adaptation strategy works well in adapting to the changing load of `Puts`. At times, as observed in the 15% and 50% `Put` phases, our adaptation algorithm fluctuates around the optimal mix of frontend and backend workers before converging to a stable mix that matches frontend producers of log records with backend gleaners that consume these log records.

Throughout the execution, for 2% `Puts`, the throughput hovers around 16 Mops, and the number of log gleaners ranges from $1-2$. This helps explain the reduction in observed throughput of bullet-full compared to the throughput of volatile in Figure 5b. After a switch to a 15% `Put` rate, the throughput switches immediately, reflecting the corresponding uptick in the gleaner count. For the 0% `Put` case, our algorithm quickly and correctly converges to a gleaner count of 0, thus explaining the throughput reported in Figure 5a that matches the throughput of volatile. For the 15% and 50% `Put` cases, the number of gleaners needed settles down to 6 and 8 respectively. Note that in our 100% `Put` experiments (not reported here in detail), we observed the number of gleaners vary between $13-15$.

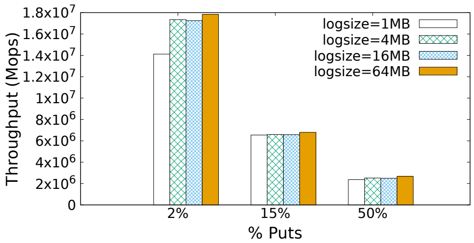


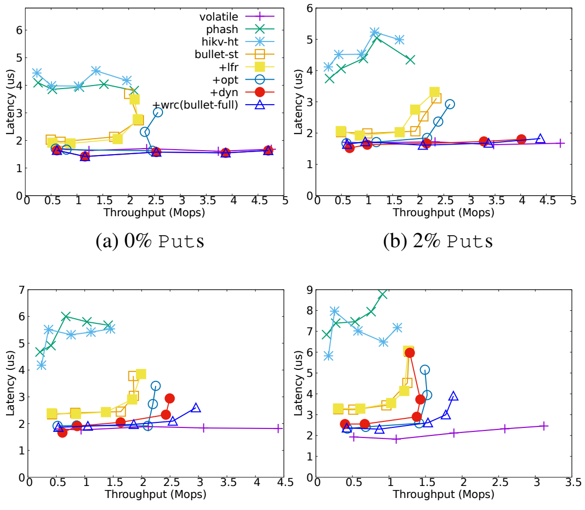
Figure 8: Effect of log size (per thread) on throughput of bullet-full.

7.4 Log Size Sensitivity

Bullet's CRLs act as speed matching buffers between the frontend and backend worker threads. As long as there is enough available space in CRLs, frontend workers keep appending log records as quickly as they can. When CRL occupancy gets too high, workers are incrementally switched to the backend to match the frontend load of CRL population. If the CRLs are too small in size, Bullet can easily enter a mode where threads bounce between frontend and backend at a high frequency, which in turn could lead to significant disruption in overall performance. The question then to consider is – how big should these logs be to avoid performance degradation due to workers switching frontend and backend roles?

To that end, Figure 8 shows the results of our experiment where we vary the per-thread log size from 64 MBs (the size we used for all experiments described above), down to 1 MB. In addition, the CRL infrastructure maintains 32 logs in-all; when a frontend worker exhausts its log, it can switch to another log that is not in use by another frontend worker. As a result, per-thread log sizes of 1, 4, 16, and 64 MBs result in total CRL footprint of 32, 128, 512, and 2048 MBs respectively. Even the largest 2048 MB CRL footprint may be acceptable in a future PM-equipped system that hosts multi-terabytes of PM.

The overall results were quite surprising to us: We expected log size to have a big impact on performance across the board. However, for write-intensive workloads, the log size does not matter to throughput. The `Put` load is high enough that the system converges to a stable mix of frontend and backend threads. The interesting case is 2% `Puts`. We observe a modest 3% drop in throughput when we transition from 64 MB logs to 4 or 16 MB logs, whereas a further reduction in log size (to 1 MB) results in a significant 20% drop in throughput. The problem with 1 MB logs is that the `Put` load generates enough log traffic to populate CRLs quickly enough that worker threads switch to the backend more aggressively than is necessary. Subsequently, a high number of of backend workers drains the log quickly after which a larger than necessary fraction of backend workers switch



(a) 0% Puts

(b) 2% Puts

(c) 15% Puts

(d) 50% Puts

Figure 9: Latency (99^{th} percentile) vs. Throughput results for test runs with independent client threads (1,2,4,6,8). The number of workers is kept to a constant 8. The graphs show the effect of increasing client load on Bullet.

to the frontend role. This over-aggressive switching of worker roles results in the performance degradation. However, 4 MB log size is big enough to absorb the log population rate more gracefully. Note that the size of each log record (including its header) is 193 bytes.

7.5 End-to-End Performance

To understand the end-to-end performance observed by independent clients, we conducted an experiment where clients generated back-to-back requests based on the zipfian distribution mentioned earlier. The clients were hosted as independent threads in Bullet's address space, eliminating the overheads related to network latencies.

Each client generates a request in its local buffer that is visible to all of Bullet's workers (but not other clients), waits for a response from Bullet, and repeats. The workers synchronize amongst each other, using a per buffer lock, to get and process client requests. We reduce contention on these locks by ensuring that workers serve a multitude of requests (1,000 in our experiments) before releasing an acquired lock and switching over to another buffer. To minimize interference between workers and client threads, we host the workers on one socket of the emulator and the client threads reside on the other socket. We effectively end up getting a maximum of 8 worker threads for each test run in this experiment.

Figure 9 shows performance of the various K-V stores with growing number of client threads. First, notice the $5X$ increase in latency of operations over all the K-V stores compared to earlier experiments (Figure 5). This slowdown was a big surprise. However, additional experimentation revealed cross-socket cache access latencies to be the biggest contributor to the overheads: when we pinned communicating workers and client threads on the same socket the latency increase reduced to approximately 10%. We did not pursue such an intermingled topological layout for clients and workers since workers tend to dynamically switch between clients when some workers are busy performing gleaning operations, which led to unpredictable performance.

Other than the unexpected NUMA effects on performance, the observed relative degradation in latencies of Bullet's flavors bullet-st, +lfr, and +opt appears to be much greater than our prior experiments (Figure 5). This degradation can be squarely attributed to the fact that these flavors of Bullet are effectively left with 4 frontend workers, and a greater number of clients (up to 8) results in overload leading to higher latencies at client counts greater than 4. Similar relative latency degradation can be observed in the 15% and 50% write loads for Bullet flavors +dyn and +wrc(bullet-full) : Some worker threads are forced to play the backend gleaner role, which increases the load on the frontend workers since the number of clients is now greater than the frontend workers.

In general, since writes are expensive, an increasing percentage of writes tends to reduce the performance gains we get from the two-tiered architecture of Bullet. We conclude that Bullet does not really close the performance gap between volatile and persistent K-V stores for write-heavy workloads. However, it significantly closes this performance gap in read-dominated workloads.

8 Conclusion

While emerging byte-addressable persistent memory technologies, such as Intel/Micron's 3D XPoint, will approach the performace of DRAM, we expect to see a non-trivial performance gap (within an order of magnitude) between them. We showed that this performance gap can have significant implications on the performance of persistent memory optimized K-V stores. In particular, we conclude that DRAM does have a critical performance role to play in the new world dominated by persistent memory. We presented our new K-V store, called Bullet, that is architected to exploit this exact observation.

We introduced *cross-referencing logs (CRLs)*, a general purpose scalable logging framework that can be used to build a two-tiered architecture for a persistent K-V store that leverages capabilities of emerging byte-addressable persistent memory technologies, and the much faster DRAM, to deliver performance approaching that of a DRAM-only K-V store for read-dominated workloads. Our performance evaluation shows the effectiveness of Bullet's architectural features that bring its performance close to that of a DRAM-only K-V store for read-heavy workloads. Write-heavy workloads' performance is severely limited by the high latency of *failure-atomic* writes, and further research is warranted to reduce these overheads.

References

[1] 3D XPoint Technology Revolutionizes Storage Memory. http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-technology-animation.html, 2015.

[2] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), pp. 53–64.

[3] Bailey, K. A., Hornyack, P., Ceze, L., Gribble, S. D., and Levy, H. M. Exploring Storage Class Memory with Key Value Stores. In *Proceedings of 1st Workshop on Interactions of NVM-Flash with Operating Systems and Workloads* (2013).

[4] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2000), ASPLOS IX, ACM, pp. 117–128.

[5] Blott, M., Karras, K., Liu, L., Vissers, K. A., Bär, J., and István, Z. Achieving 10gbps line-rate key-value stores with fpgas. In *5th USENIX Workshop on Hot Topics in Cloud Computing* (2013).

[6] Bridge, B. Nvm-direct library. *https://github.com/oracle/nvm-direct*, 2015.

[7] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems 26*, 2 (June 2008), 4:1–4:26.

[8] Chen, S., and Jin, Q. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment 8*, 7 (2015), 786–797.

[9] Chi, P., Lee, W., and Xie, Y. Making b$^+$-tree efficient in pcm-based main memory. In *International Symposium on Low Power Electronics and Design, ISLPED'14, La Jolla, CA, USA - August 11 - 13, 2014* (2014), pp. 69–74.

[10] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), pp. 105–118.

[11] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), pp. 143–154.

[12] Debnath, B., Haghdoost, A., Kadav, A., Khatib, M. G., and Ungureanu, C. Revisiting hash table design for phase change memory. *SIGOPS Oper. Syst. Rev. 49*, 2 (Jan. 2016), 18–26.

[13] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007* (2007), pp. 205–220.

[14] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), pp. 401–414.

[15] Fan, B., Andersen, D. G., and Kaminsky, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), pp. 371–384.

[16] Giles, E., Doshi, K., and Varman, P. J. Softwrap: A lightweight framework for transactional support of storage class memory. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015* (2015), pp. 1–14.

[17] Harris, T. L. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings* (2001), pp. 300–314.

[18] Apache HBase. *http://hbase.apache.org/*.

[19] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing* (2003), pp. 92–101.

[20] Hetherington, T. H., O'Connor, M., and Aamodt, T. M. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), pp. 43–57.

[21] Hosomi, M., Yamagishi, H., Yamamoto, T., Bessho, K., Higo, Y., Yamane, K., Yamada, H., Shoji, M., Hachino, H., Fukumoto, C., Nagao, H., and Kano, H. A novel non-volatile memory with spin torque transfer magnetization switching: Spin-RAM. *International Electron Devices Meeting* (2005), 459–462.

[22] Hu, W., Li, G., Ni, J., Sun, D., and Tan, K.-L. BP-Tree : A Predictive B+-Tree for Reducing Writes on Phase Change Memory. *IEEE Transactions on Knowledge and Data Engineering 26* (2014), 2368–2381.

[23] Huai, Y. Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects. *AAPPS Bulletin 18*, 6 (2008), 33–40.

[24] Intel® 64 and IA-32 Architectures Software Developer's Manual. *http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf*, 2015.

[25] Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., and Ailamaki, A. Aether: A Scalable Approach to Logging. *Proceedings of VLDB Endowment 3*, 1-2 (Sept. 2010), 681–692.

[26] Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., and Ailamaki, A. Scalability of Write-ahead Logging on Multicore and Multisocket Hardware. *The VLDB Journal 21*, 2 (Apr. 2012), 239–263.

[27] Kalia, A., Kaminsky, M., and Andersen, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), pp. 295–306.

[28] Lakshman, A., and Malik, P. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[29] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[30] LevelDB – a fast and lightweight key/value database library. *http://leveldb.org/*.

[31] Li, S., Lim, H., Lee, V. W., Ahn, J. H., Kalia, A., Kaminsky, M., Andersen, D. G., Seongil, O., Lee, S., and Dubey, P. Architecting to achieve a billion requests per second

throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (2015), pp. 476–488.

[32] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), pp. 429–444.

[33] Lim, K., Meisner, D., Saidi, A. G., Ranganathan, P., and Wenisch, T. F. Thin servers with smart pipes: Designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), pp. 36–47.

[34] Mao, Y., Kohler, E., and Morris, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), pp. 183–196.

[35] Marathe, V. J., Mishra, A., Trivedi, A., Huang, Y., Zaghloul, F., Kashyap, S., Seltzer, M., Harris, T., Byan, S., Bridge, B., and Dice, D. Persistent Memory Transactions *https://arxiv.org/abs/1804.00701*, 2018.

[36] Memcached – a distributed memory object caching system. *https://memcached.org/*.

[37] Mitchell, C., Geng, Y., and Li, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (2013), pp. 103–114.

[38] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems 17*, 1 (1992), 94–162.

[39] Nawab, F., Izraelevitz, J., Kelly, T., Morrey, C. B., Chakrabarti, D., and Scott, M. L. Dali: A Periodically Persistent Hash Map. In *Proceedings of the 31st International Synposium on Distributed Computing* (2017).

[40] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), pp. 385–398.

[41] Oukid, I., Lasperas, J., Nica, A., Willhalm, T., and Lehner, W. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 371–386.

[42] Pelley, S., Chen, P. M., and Wenisch, T. F. Memory persistency. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014* (2014), pp. 265–276.

[43] Rao, D. S., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System software for persistent memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014* (2014), p. 15.

[44] Redis – in-memory data structure store, *http://redis.io/*.

[45] Rudoff, A. Deprecating the PCOMMIT Instruction. *https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction*, 2016.

[46] Strukov, D. B., Snider, G. S., Stewart, D. R., and Williams, R. S. The missing Memristor found. *Nature 453* (2008), 80–83.

[47] Suzuki, K., and Swanson, S. A Survey of Trends in Non-Volatile Memory Technologies: 2000-2014. In *2015 IEEE International Memory Workshop* (2015), pp. 1–4.

[48] Swift Object Store. *https://swift.openstack.org/*.

[49] The SNIA NVM Programming Technical Working Group. NVM Programming Model (Version 1.0.0 Revision 10), Working Draft. http://snia.org/sites/default/files/NVMProgrammingModel_v1r10DRAFT.pdf, 2013.

[50] Volos, H., Tack, A. J., and Swift, M. M. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), pp. 91–104.

[51] Wang, P., Sun, G., Jiang, S., Ouyang, J., Lin, S., Zhang, C., and Cong, J. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), pp. 16:1–16:14.

[52] Wang, T., and Johnson, R. Scalable logging through emerging non-volatile memory. *PVLDB 7*, 10 (2014), 865–876.

[53] Wu, X., Zhang, L., Wang, Y., Ren, Y., Hack, M., and Jiang, S. zexpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), pp. 14:1–14:15.

[54] Xia, F., Jiang, D., Xiong, J., and Sun, N. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference* (2017), pp. 349–362.

[55] Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K. L., and He, B. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (2015), pp. 167–181.

[56] Zhang, K., Wang, K., Yuan, Y., Guo, L., Lee, R., and Zhang, X. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1226–1237.

[57] Zhang, Y., and Swanson, S. A study of application performance with non-volatile main memory. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015* (2015), pp. 1–10.

[58] Zhou, J., Shen, Y., Li, S., and Huang, L. NVHT: An Efficient Key-value Storage Library for Non-volatile Memory. In *Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies* (2016), pp. 227–236.

Metis: Robustly Optimizing Tail Latencies of Cloud Systems

Zhao Lucis Li[⋆‡] Chieh-Jan Mike Liang[‡] Wenjia He[⋆‡] Lianjie Zhu[○] Wenjun Dai[○]
Jin Jiang[○] Guangzhong Sun[⋆]
[⋆]*USTC* [‡]*Microsoft Research* [○]*Microsoft Bing Ads*

Abstract

Tuning configurations is essential for operating modern cloud systems, but the difficulty arises from the cloud system's diverse workloads, large system scale, and vast parameter space. Building on previous space exploration efforts of searching for the optimal system configuration, we argue that cloud systems introduce challenges to the robustness of auto-tuning. First, performance metrics such as tail latencies can be sensitive to non-trivial noises. Second, while treating target systems as a black box promotes applicability, it complicates the goal of balancing exploitation and exploration. To this end, Metis is an auto-tuning service used by several Microsoft services, and it implements customized Bayesian optimization to robustly improve auto-tuning: (1) diagnostic models to find potential data outliers for re-sampling, and (2) a mixture of acquisition functions to balance exploitation, exploration and re-sampling. This paper uses Bing Ads key-value store clusters as the running example – compared to weeks of manual tuning by human experts, production results show that Metis reduces the overall tuning time by 98.41%, while reducing the 99-percentile latency by another 3.43%.

1 Introduction

For many web-scale cloud systems, main evaluation metric is tail latencies (e.g., 99 and 99.9-percentile latencies) of serving a request [9]. While tail latencies seem rare, the probability of a user request experiencing the tail latency in an end-to-end system can be high, especially that most web-scale applications employ a multi-stage architecture. Imagine a web-scale application with a five-stage processing pipeline, the probability of a request encountering 99-percentile latency at least is $\sim$5%. Furthermore, tail latencies can be more than 10 times higher, as compared to the average latency [9].

Recently, advances in machine learning have spawned strong interests in automating system customizations, where auto-tuning system configuration of parameters is a popular scenario [2, 3, 20]. Notably, cloud systems exhibit two motivating characteristics for auto-tuning. First, the overhead of operating cloud systems is increasingly larger, due to the increasingly more dynamic and variable system workloads, and the scale of modern cloud systems. In many read-intensive web applications such as news sites and advertisement networks, data queries are tied to end-user personal interests and current contexts, which can exhibit temporal dynamics. Furthermore, even within one cloud system, there can be several components that individually impose different data requirement and handle different types of meta data and user data. Second, as cloud systems become more complicated, both design space and parameter space grow significantly. The optimal system configuration is beyond what human operators can efficiently and effectively reason about, and the cost to naïvely benchmark all possible system configurations is exorbitant.

Bayesian optimization (BO) with Gaussian process (GP) has emerged as a powerful black-box optimization framework for system customizations [2, 3, 19]. Mathematically, we model the configuration-vs-performance space by regressing over data points already collected, i.e., system configurations benchmarked. And, this regression model allows us to estimate the global optimum, or the best-performing system configuration. While collecting more data points can improve the regression model, benchmarking one system configuration of parameters can be time-consuming due to system warm-up and stabilization. Fortunately, BO offers a way to iteratively build up the training data by suggesting system configurations to benchmark, with the goal to maximally

Chieh-Jan Mike Liang is the corresponding author. This work was done when Zhao Lucis Li and Wenjia He were interns at Microsoft Research.

improve the regression model accuracy.

1.1 Contributions

In this paper, we report the design, implementation, and deployment of *Metis*, an auto-tuning service used by several Microsoft services for robust system tuning. While Metis is inspired by previous efforts to leverage BO to train the GP regression model, we address the following factors to improve the robustness of optimizing system customizations.

First, since the GP regression model is trained with data points from benchmarking some system configurations, how well these data points capture the configuration-vs-performance space's global optimum determines the auto-tuning effectiveness. At the same time, we should avoid unnecessarily over-sampling the space, as system benchmarking can be resource-intensive and time-consuming. To this end, at each iteration, BO's strategy for picking the next system configuration to benchmark should balance *exploitation* (i.e., regions with high probability of containing optimum) and *exploration* (i.e., regions with high uncertainty of containing optimum). Specifically, inadequate exploration reduces the chance of moving away from a local optimum, and inadequate exploitation impacts the efficiency of identifying the global optimum. In contrast to related efforts that rely on simple-to-implement strategies [2, 3], Metis strongly decouples exploitation and exploration, so that it can independently evaluate each action's expected improvement and anticipated regret.

Second, the theory behind BO and GP mostly assumes the data collection is reasonably noise-free, or susceptible to only the Gaussian noise. Unfortunately, many system performance metrics (e.g., tail latencies) are sensitive to non-Gaussian or unstructured noise sources in the wild, e.g., CPU scheduling and OS updates. In contrast to related efforts [2, 3], not only does Metis consider exploitation and exploration, but it also weighs the benefit of *re-sampling* existing data points. Key enabler is the diagnostic model that Metis creates to identify potential outliers.

To demonstrate the usefulness of Metis for real-world system black-box optimizations, this paper showcases one of our production deployments as the running example – optimizing tail latencies of Microsoft Bing Ads key-value (KV) stores. We present measurements and experiences from two KV clusters that handle $\sim$4.14 billion and $\sim$3.18 billion key-value queries per day, respectively. Compared to weeks of manual tuning by human experts, production results show that Metis reduces the overall tuning time by 98.41%, while reducing the 99-percentile latency by another 3.43%.

2 Background and Motivation

As a black-box optimization service, Metis tunes the configuration of several Microsoft services and networking infrastructure. This section describes one deployment as the running example in this paper – Microsoft Bing Ads key-value (KV) store cluster, *BingKV*. Then, we motivate optimally customizing cloud systems.

2.1 Running Example: *BingKV*

Individual stages of the ads query processing pipeline, e.g., selection and ranking, host separate KV store clusters. Within the cluster, each KV server is responsible for one non-overlapping dataset partition. Therefore, each server can experience different workload, as defined by the frequency distribution and the size distribution of top queried key-value objects. Manually tuning each server is difficult as a cluster can have thousands of servers.

The configuration space consists of parameters of the local caching mechanism, which decides what KV objects should be cached in the in-memory cache or served from the persistent data store. Main evaluation metric is tail latencies of serving key queries after the in-memory cache is full. We describe these parameters and their typical value ranges next.

First, both the recency and frequency distribution are well-known foundation for cache eviction. BingKV supports `NumCacheLevels` (1 - 10) levels of LRU (Least Recently Used) or LFU (Least Frequently Used) based caches – a cached object can move up a level if it has been queried `CachePromotionThreshold` (1 - 1,000) times. Since cached objects at higher levels have been queried more than those at lower levels, locality principle implies that they should not be easily evicted. Therefore, we allow `NumInevictableLevels` (0 - 9) levels to be specified as being inevictable.

Second, many cache designs incorporate a shadow buffer that stores the key only, instead of the entire key-value object. BingKV implements a shadow buffer with a capacity of `ShadowCapacity` (1 - 10) MB, to hold keys with an object size larger than `AdmissionThreshold` (1 - 1,000) bytes. Then, keys in the shadow buffer are moved to the cache only if they have been queried more than `ShadowPromotionFreq` (1 - 1,000) times.

Finally, `CacheCapacity` (1 - 10,000) specifies the cache capacity (excluding the shadow buffer) in MB, and the dataset partition on each server is divided into `NumShards` (1 - 64) shards.

2.2 Impacts of Suboptimal Configurations

To illustrate impacts of suboptimal configurations, we empirically measure BingKV's 99-percentile latency un-

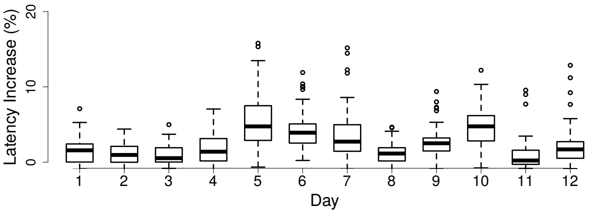


Figure 1: Motivating example – suboptimal configurations can impact the system performance. We use BingKV as the running example, and compare the tail latency increase with respect to the optimal system configuration.

der different parameter configurations, and quantify the tail latency increase with respect to the best-performing system configuration. Experiments use two 12-day pre-recorded production workload traces of BingKV, *Prod_Trace*$_1$ and *Prod_Trace*$_2$, and replay these workload traces to benchmark system configurations.

Suboptimal configurations can happen when system tuning does not consider machine-specific setup. To illustrate, we vary the cache capacity between 32 MB and 512 MB – we find the best-performing configuration (e.g., `AdmissionThreshold` ranging from 1 to 1,000) for each capacity, and then we compare the latency of the best-performing configuration to 100 configurations uniformly sampled from the parameter space. Results show that the average 99-percentile latency increase can range from 15.34% (for 256-MB cache) to 22.74% (for 512-MB cache), with more than 34.14% increase in the worst case. For *Prod_Trace*$_2$, the average 99-percentile latency increase can range from 10.21% (for 32-MB cache) to 13.89% (for 64-MB cache), with more than 13% increase in the worst case.

Suboptimal configurations can also happen when system tuning does not consider temporal dynamics. For each day of *Prod_Trace*$_1$, we compare the best-performing configuration to that of the other 11 days. Figure 1 shows that the average 99-percentile latency increase can be as high as more than 5.58% (day 5).

2.3 Strawman Solutions

Manual tuning. Manual tuning can leverage administrators' knowledge and prior experience about the target system. Unfortunately, as modern systems get larger and more complicated, reasoning the system behavior in a high-dimensional configuration space becomes increasingly difficult. Furthermore, real-world workload can have dynamics across machines and time, and manual tuning does not scale. To illustrate this argument, our experience shows that manually tuning a subset of BingKV can take weeks. And, in some cases, it is not certain whether the manual tuned configuration is indeed the best-performing one.

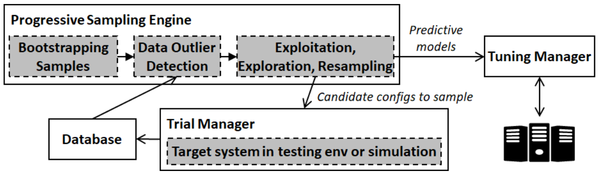

Progressive Sampling Engine
Bootstrapping Samples
Data Outlier Detection
Exploitation, Exploration, Resampling
Predictive models
Tuning Manager
Candidate configs to sample
Database
Trial Manager
Target system in testing env or simulation

Figure 2: Architecture of Metis service.

Naïve Bayesian optimization. OtterTune [2] and CherryPick [3] demonstrated the potential of Bayesian optimization in adaptively finding the best-performing configuration for databases and data analytics systems, respectively. Both approaches adopt common BO selection strategies such as Expected-Improvement (EI). While these strategies are easy-to-implement, they do not consider factors impacting the robustness of tuning system customizations: the balance of exploitation and exploration [14], and data outliers.

3 Our Approach

Given a workload of a system, Metis has the objective of predicting the best-performing configuration by selectively exploring the configuration-vs-performance space. Through system benchmarking, Metis can collect data points that describe system inputs (e.g., system workload and parameter values) and outputs (e.g., performance metrics of interests) for training its model. When Metis does not change its prediction of the best-performing configuration with additional data points, we say the model has converged. Maximizing the prediction accuracy and minimizing the model convergence time are two evaluation metrics for Metis. The former relates to the auto-tuning correctness, and the latter relates to the service scalability.

This section first outlines the usage flow of Metis. Then, we formulate auto-tuning as an optimization problem, and highlight practical challenges to motivate customizations proposed in subsequent sections.

3.1 Architectural Overview

Figure 2 shows the architecture of Metis containing components for model training and system tuning.

Model training. The Progressive Sampling Engine solves the optimization problem of robustly constructing the predictive regression model, which models the configuration-vs-performance space. Each model corresponds to one performance metric, and the model dimensionality depends on the number of parameters.

Users start by providing the Progressive Sampling Engine their requirements (e.g., parameters and metric of interest) and workload traces (e.g., production or synthesized traces). The engine bootstraps BO by picking a set of system configurations as bootstrapping trials, for the Trial Manager to benchmark. Benchmarking can happen in simulators or real machines. Then, using performance metrics collected as the training data points, the engine picks subsequent system configuration of parameters to benchmark. Ideally, each iteration should improve the regression model's estimation of the space's global optimum. We formulate this iterative process as an optimization problem in the next subsection (c.f. §3.2).

When a stopping criterion is met, the Progressive Sampling Engine stops collecting more data points, and it outputs the GP regression model trained so far. Stopping criteria can include training time budget and so on. As we show in §6, the system benchmarking time dominates training, rather than the model computation time.

System tuning. The Tuning Manager periodically receives status reports from individual servers of the target system. These status reports contain current workload characterizations and logged performance metrics. If performance degradation is detected (e.g., a significant increase in the tail latency), the Tuning Manager uses the workload characterization to select the nearest regression model. We use DNNs to classify workloads, which minimizes the burden of weighing workload features for classification.

3.2 Optimization Problem Formulation

Formally, for a workload w, we want to find the k-dimensional configuration c^* (representing k system parameters), which has the highest probability of being the best-performing configuration, c_w. The *overall problem objective* is as follows:

$$c^* = \underset{c \,\in\, configs}{\operatorname{argmax}} \; P(c = c_w \mid w)$$

Given the k-dimensional parameter space can be too large for exhaustive searches, Metis opts the regression model to predict with limited amount of data points collected. While more training data will help reducing the regression uncertainty, the *training objective* should also consider the training overhead:

$$\begin{aligned} \text{minimize} \sum_{w \,\in\, workloads} & (confidence\_interval(c_w)) \\ \text{subject to} & \sum T(c) \;\leq\; T_{budget} \end{aligned}$$

$T(c)$ denotes the time necessary to sample a configuration c, and T_{budget} denotes the maximum amount of time cost allocated for model training.

Predictive regression model formulation. Regression allows Metis to model the expected configuration-vs-performance space, with data points already collected from benchmarking some configurations. The regression model captures the conditional distribution of a target performance metric given a system configuration. We pick Gaussian process (GP) [13] as the model, which extends multivariate Gaussian distributions to infinite dimensionality, such that observations for an unsampled data point are assumed to follow a multivariate Gaussian distribution. In other words, assuming a stochastic function f where every input x has an output $f(x)$, each $f(x)$ is defined as a mean μ and a standard deviation σ of a Gaussian distribution.

GP exhibits a number of desirable properties. First, it does not assume a certain mathematical relationship between model inputs and outputs, e.g., the linear relationship. Second, for any x, GP can return the expected value of $f(x)$ and uncertainty (i.e., standard deviation).

Optimization framework. A proven approach to train the GP model is *Bayesian optimization* (BO) [17]. At each iteration, based on the current GP model, BO selects a system configuration to benchmark next to further train the GP model. The selected configuration is expected to maximally improve the accuracy predicting the global optimum of the configuration-vs-performance space. The logic behind selecting the next trial is implemented by BO's acquisition function, and its design traditionally aims to balance exploitation (i.e., sampling regions with high probability of containing optimum) and exploration (i.e., sampling regions with high uncertainty). The model converges when BO believes that the GP model has sufficiently captured the configuration-vs-performance space global optimum, or the best-performing system configuration in our case.

BO conceptually realizes a form of progressive sampling, and it is suitable for scenarios where collecting system performance metrics of a single trial is resource-intensive or time-consuming (e.g., waiting for a system to warm up and stabilize).

3.3 Practical Challenges of Robustness

Running BO with GP requires the following practical considerations for robust system tuning.

Sampling configuration-vs-performance space without strong assumptions on the target system behavior. Given that system behavior can be difficult to be properly reasoned, Metis tries to learn the configuration-vs-performance space from selective sampling. In other words, based on system configurations already benchmarked, Metis selects the next system configuration to benchmark, which is expected to maximally improve the regression accuracy. To this end, we believe that existing efforts leave the following two gaps to fill.

First, balancing exploitation and exploration is still a non-trivial problem. Many proposed acquisition functions evaluate the potential improvement of a trial [19], and the community has shown their limitations in balancing exploitation and exploration [4, 17]. Expected Improvement (EI) is a widely popular choice of acquisition functions. It improves upon Maximum Probability of Improvement, by estimating the best-case improvement with both the mean and the uncertainty around a sampling point. However, Ryzhov et al. [14] showed that EI allocates only O(log n) samples to regions that are expected to be suboptimal, where n is the total number of trials. In other words, n needs to be significantly large for EI to balance exploitation and exploration.

Second, bootstrapping trials refer to the set of sampling points to initialize BO. Since BO decides the next trial with the GP model trained with past trial data points, prior data samples can influence how BO expects the posterior expectation to be. The main requirement for selecting bootstrapping trials should be exploration. Random sampling is a simple approach to pick bootstrapping trials, but it requires a large amount of sampling points to ensure coverage [12]. This is not ideal for time-consuming trials, where some systems need time to stabilize (e.g., system cache warm-up).

Handling non-Gaussian data noise. Most theoretical work on BO with GP assumes the trial data collected are noise-free (i.e., perfectly reproducible experiments), or susceptible to only the Gaussian noise [17]. Unfortunately, many system metrics such as tail latencies are sensitive to a wide range of noise sources in the real world, ranging from background daemons, local resource sharing, networking variability, etc. This is different from reproducible metrics such as classification accuracy. Since real-world data noise sources can be heterogeneous and non-trivial to model, the auto-tuning service should consider the benefits of removing potential outliers.

4 Improving Auto-Tuning Robustness

Given the challenges discussed in §3.3, we now discuss our customizations to Bayesian optimization to improve tuning robustness.

4.1 Bootstrapping Trials

To select sampling points to bootstrap Bayesian optimization for a given workload, Metis exercises Latin Hypercube Sampling (LHS) [12]. LHS is a type of stratified sampling. According to some assumed probability distribution, LHS divides the range of each of M parameters into I equally probable intervals, and randomly selects only *one* single data point from each interval. Since each interval of each parameter is selected only once, the number of bootstrapping trials chosen by LHS is exactly I, regardless of M. Furthermore, the maximum number of possible combinations for bootstrapping trials is only $(\prod_{i=0}^{I-1} I-i)^{M-1} = (I!)^{M-1}$.

LHS offers several desirable properties for bootstrapping Bayesian optimization. First, it has been shown that, compared to random sampling, stratified sampling can reduce the sample size required to achieve the same conclusion [12]. Second, compared to another well-known stratified sampling technique, Monte Carlo, LHS allows one to obtain a stable output with less samples [7]. And, while quasi-Monte Carlo can be an alternative approach, its output is a low discrepancy sequence which is not random, but uniformly deterministic [6]. Third, as LHS does not control the sampling of combinations of dimensions, the number of samples picked LHS is agnostic to the dimensionality of the parameter space.

A well-known concern of LHS is that it may exhibit a higher memory consumption, in order to keep track of parameter intervals that have already been sampled. However, we note that only a few sampling points are necessary to bootstrap Metis, and our current running system sets I to be 5.

4.2 Customized Acquisition Function

After obtaining bootstrapping sampling trials with LHS, Metis then runs Bayesian optimization to iteratively train the Gaussian process model. At each iteration, BO outputs the system configuration that is expected to offer the most improvement to the GP model, in terms of predicting the best-performing system configuration. This output represents the system configuration that Metis should sample in the next iteration of training. To produce this output, Metis customizes the acquisition function to balance three goals: *exploitation*, *exploration*, and *re-sampling*.

Our customized acquisition function works as a mixture of three separate sub-acquisition functions (c.f. §4.2.1) corresponding to each goal. At each iteration of BO, based on the GP model constructed so far, individual sub-acquisition functions compute the next system configuration to sample to maximize their own goal. Then, the acquisition function evaluates these candidates in terms of the improvement of the expected best-performing system configuration, and selects the candidate with the highest gain to sample next (c.f. §4.2.2).

Compared to related efforts, our acquisition function exhibits two differences. First, by introducing re-sampling as a possible action, we allow Metis to re-sample a trial whose previous results might be susceptible to non-Gaussian noise. Second, Metis decouples all three actions and runs separate acquisition functions for each action. This decoupling allows Metis to independently evaluate the potential information gain and regret aversion from taking each action in the next iteration.

4.2.1 Sub-acquisition functions

Metis decouples exploitation, exploration, and re-sampling into three separate sub-acquisition functions. Given past trials, each sub-acquisition function outputs a system configuration to maximize its own goal. These system configurations become candidates for the acquisition function.

Exploration. This sub-acquisition function looks for the sampling point whose expected observation would have with the highest uncertainty. Formally, in the GP regression model, this sampling point would exhibit the largest confidence interval.

Exploitation. This sub-acquisition function looks for the sampling point whose expected observation would most likely be the system optimum. While one simple approach is to consider the absolute observation of a sampling point as expected by the GP regression model, its accuracy can be impacted by GP's uncertainty of that sampling point. Therefore, Metis takes an approach inspired by TPE [5], and tries to estimate the probability of a sampling point giving near-optimum observation.

The exploitation sub-acquisition function works as follows. It first separates the past trials into two groups: the first group has best observations, and the second group has the rest. Then, we construct two Gaussian mixture models (GMM) to describe each group. With these two GMMs, we can evaluate the likelihood of a system configuration, c, being in either group, i.e., $P_1(c)$ and $P_2(c)$. The ratio of these two likelihood, or $\frac{P_1(c)}{P_2(c)}$, would give us a score of how likely a sampling point is in the first group, instead of second group.

Re-sampling. This sub-acquisition function looks for outliers in past trials, and suggest trials that should be re-sampled (i.e., system configurations that should be re-benchmarked).

The re-sampling sub-acquisition function works by creating the diagnostic model. To examine a data point, the diagnostic model quantifies the difference between the measured value and the expected value. To do so, the diagnostic model is a GP regression model trained without the examined data point. Then, the diagnostic model can calculate the expected value and 95% confidence interval. If the measured value falls outside the confidence interval, it is probable that the examined data point could be an outlier. The sub-acquisition function repeats the evaluation for all past trials, and it selects the trial that is farthest from the expected confidence interval.

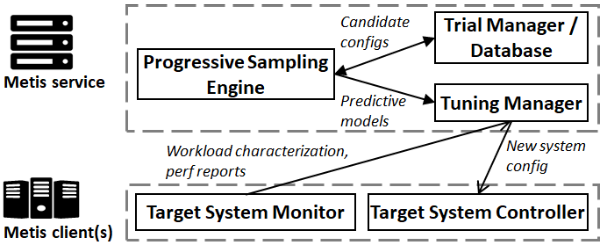


Figure 3: Components of Metis service implementation.

4.2.2 Evaluation of Candidates

Taking sub-acquisition function outputs as candidate configurations, our acquisition function computes their information gain with respect to how the prediction of optimal configuration changes. Conceptually, for each candidate, the GP model bounds the expected observation with a confidence interval. And, the upper and lower bound of this interval can help us estimate the potential information gain (if we were to actually select the corresponding candidate for benchmark).

The acquisition function starts by predicting the currently best-performing system configuration, c_{cur_best}, and this step searches for the sampling point with the lowest expected observation in the GP regression model. Then, to evaluate a candidate, $c_{candidate}$, we add its lower-bound confidence interval (i.e., best-case) to the GP regression model, and predict the would-be best-performing system configuration, c_{new_best}. The different between the expected observation of c_{cur_best} and c_{new_best} is the improvement, and 0 if the improvement is negative. Then, we repeat the process for upper-bound confidence interval (i.e., worst-case).

Finally, the acquisition function outputs the $c_{candidate}$ with the highest sum of improvement calculated with the lower-bound and the upper-bound confidence interval.

5 Implementation

Figure 3 illustrates the Metis system components implemented. Our current implementation is in Python, and this language choice allows us to use the popular Scikit-learn library for Gaussian process regression [16]. We choose the summation of Matern (3/2) and white noise as the covariance kernel [19].

Separation of logic and execution. Metis consists of a centralized web service and client stubs that sit on servers hosting the target system. Network communications happen over TCP, and the message format is JSON. Conceptually, the client stub hooks up to the target system, and it abstracts away system-specific interfaces and execution from the web services. While an alternative implementation is to place both the logic and execution in a local service on each server, the separation provides several benefits. First, the computation-intensive logic of Metis does not contend for resources on production servers. Second, being centralized, Metis has the global view of all status reports from all servers. This allows Metis to continuously improve the predictive model with new data points.

Metis web service. The web service trains one predictive regression model for each workload trace. And, these traces can be recorded in the production environment, or synthesized with tools such as YCSB [8]. Trial Manager replays each workload trace in simulators or real servers, and each replay represents one trial benchmarking one selected system configuration. Then, the training dataset consists of system configurations and corresponding performance benchmarks. In the case of BingKV, we built a simulator wrapping the production KV code binaries, to run trials.

From status reports uploaded by client stubs, if Tuning Manager detects that a server is experiencing a performance degradation above some user-specified thresholds (e.g., tail latencies have increased by more than 10% in the last six hours), it computes a new configuration and sends a command to the server's client stub.

Metis client stubs. The client stub deals with system-specific interfaces, ***(1)*** to periodically upload recent system performance measurements and workload characterization, and ***(2)*** to execute configuration change commands from the web service.

Most web-scale cloud systems already have an extensive logging mechanism for continuous performance monitoring, and Metis client stubs periodically retrieve relevant performance measurements from the same logging mechanism. While different systems have different workload characterization features, these features are either already available in the logging mechanism, or require additional code instrumentation. In the case of BingKV, our workload features are the size and query frequency of the top queried KV objects, and the incoming query traffic rate.

Upon receiving a configuration change command, Metis does not try to aggressively re-configure servers. Instead, it employs a guard time following a reconfiguration to allow the target system to warm up and stabilize. Other than time durations, this guard time can also be values of system states, e.g., cache occupancy.

	Selection of Configs	Modeling	DO
Random	Random		No
iTuned [10]	LHS + EI	BO w/ GP	No
CherryPick [3]	Random + EI	BO w/ GP	No
TPE [5]	Random + EI	GMM	No
Metis	LHS + Customized	BO w/ GP	Yes

Table 1: This table highlights differences among comparison baselines and Metis. The ”DO” column shows whether data outlier removal is considered.

6 Evaluation

Our major results include – ***(1)*** compared to baselines, Metis picks better configurations 84.67% of times in the presence of low data noises. ***(2)*** In the presence of data outliers, Metis has 56.77% more chance of picking better configuration. ***(3)*** Metis has a faster convergence time in searching for the best-performing configuration.

6.1 Methodology

We use the Bing Ads KV store (c.f. §2) as the target system. Our testbed machines have two 2.1 GHz CPUs (with 16 cores), 16 GB RAM, and a 256 GB SSD. These machines host the target system binaries, and simulate configurable key-value query arrival rates (or queries per second) for the given workload trace. We log performance metrics with asynchronous I/Os to minimize any artifact introduced by logging.

Datasets and workload traces. We obtained two 12-day key-value query traces, $Prod\_Trace_1$ and $Prod\_Trace_2$, from two Bing Ads KV store clusters, $BingKV_1$ and $BingKV_2$. These two real-world traces exhibit following characteristics – the average size of top 500 frequently queried KV objects can have a difference up to 9.49% from one week to another, and that of $Prod\_Trace_1$ is approximately two-times larger than $Prod\_Trace_2$.

For more controlled experiments, we also synthesized three workload traces containing ∼0.5 million query requests. Our workload generation tool is based on the Yahoo! Cloud Serving Benchmark (YCSB) [8]. However, while YCSB considers the key-value frequency distribution, it does not consider the key-value size distribution. To fill this gap, given a list of unique keys from YCSB (sorted in descending order of frequency), we assign key-value sizes according to some distributions. For $Synth\_Trace_1$, $Synth\_Trace_2$ and $Synth\_Trace_3$, we fix their size distribution to be Zipf, and we generate keys following the frequency distribution of Zipf, normal, and linear. Doing so allows us to examine different size distributions.

Comparison baselines. In addition to random search,

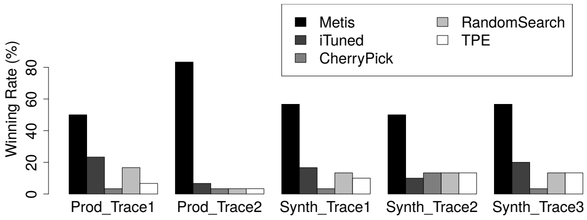


Figure 4: Chance that each approach picks the best configuration. Each experiment allows each approach to run 25 trials, and we repeat the experiment 30 times.

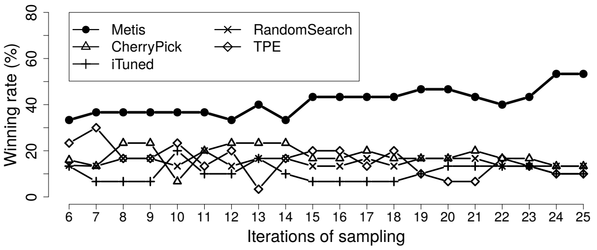


Figure 5: Chance that each approach picks the best configuration for $\mathit{Prod\_Trace}_1$, while changing the number of sampling points allowed.

we take state-of-the-art BO-driven approaches including iTuned [10], CherryPick [3], and TPE [5]. We configure these comparison baselines with the recommended setting, e.g., the Matern(5/2) kernel for CherryPick. At each iteration, these approaches output the configuration they expect to have lowest tail latency. Table 1 lists their differences. For the ground truth, we brute-force all possible configurations for $\mathit{Prod\_Trace}_1$.

6.2 Effectiveness of Metis

This section quantifies the likelihood that the system configuration selected by Metis outperforms those of other approaches. This relates to the auto-tuning correctness. We also discuss factors including time budgets (i.e., number of trials allowed) and data outliers.

Metis has a higher chance in picking system configurations that outperforms those of other approaches. Assuming a fixed time budget, we allow each approach to run 25 trials. We then rank approaches by the 99-percentile latency of their expectedly best-performing system configurations. We clean-install the machine to ensure minimal noise in measurements, and evaluate the case of data outliers later in this section. Experiments are repeated 30 times, to compensate for the randomness in some approaches. For random search, we take the best of all 25 trials for evaluation. We note that TPE requires at least 20 bootstrapping trials (from random sampling), and we allocate five bootstrapping trials to CherryPick, iTuned and Metis.

Figure 4 summarizes results when system benchmarks, or training data, are relatively noise-free. It shows that Metis has a higher chance of picking the winning configuration, i.e., the system configuration that outperforms those selected by comparison baselines. Depending on the synthesized workload trace, Metis can outperform 48.97% (for $\mathit{Synth\_Trace}_2$) to 84.67% (for $\mathit{Prod\_Trace}_2$) of times.

Furthermore, taking a closer look into cases where Metis failed to win, we note that Metis 99-percentile latency is within 1% of the winner. In addition, Metis observes the lowest difference of 99-percentile latency to the optimal, on average. This difference is 0.44% for Metis, which is 67.16% lower than CherryPick.

Metis has a faster convergence time than other approaches. Regardless of the parameter tuning approach, the effectiveness in finding the optimal configuration should ideally increase with the number of data points already sampled, or configurations benchmarked. One natural question is how Metis converges to the optimal system configuration, as compared to other approaches. To this end, we try to answer two questions: ***(1)*** at each iteration, what is the likelihood that Metis selects the system configuration that outperforms those of other approaches? ***(2)*** how fast does Metis converge to the optimal system configuration?

Figure 5 shows that the chance for Metis to pick the winning configurations is high at all iteration. We repeat this experiment 30 times, due to the randomness in some approaches. This result is desirable for auto-tuning under limited time budget, as system benchmarks can consume a long time. In addition, the figure shows that, as the number of sampled data points increase, Metis is able to better model the configuration-vs-performance space, which increases the likelihood of picking the winning configuration. Furthermore, looking at configurations that CherryPick and iTuned pick, we observe that they lean towards exploitation. On the other hand, Metis independently evaluates the potential information gain from exploitation and exploration.

Following Figure 5, Figure 6 illustrates how each approach converges to the global optimum of the configuration-vs-performance space. The figure calculates the average 99-percentile latency of selected configurations from 30 repeated experiments. Compared to other approaches, Metis has a faster convergence time, as it is able to benchmark system configurations that would iteratively improve modeling the configuration-vs-performance space. We note that the effectiveness of TPE significantly improves after 20 iterations, as its Gaussian Mixture Models require many training data. Fi-

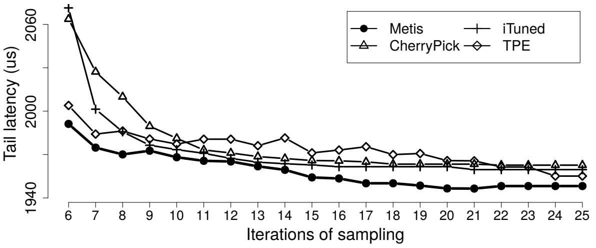


Figure 6: The search path visualizes the tail latency of system configurations that BO-driven approaches select at each iteration.

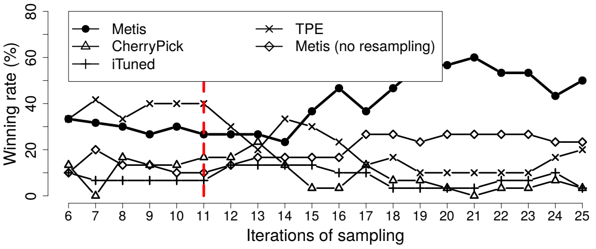


Figure 7: Chance that each approach picks the winning configuration for $Prod\_Trace_1$. The test machine has intermittent background activities that result in data outliers. The red line marks when Metis has sufficient data points for re-sampling.

nally, we also note that Figure 6 suggests that iTuned and CherryPick have a higher chance of exploiting local optimum, and thus their acquisition function would require more iterations to find the global optimum.

Metis maintains high chance of picking the winning configuration in the presence of data outliers. Building on the discussion so far, we now demonstrate the robustness of Metis to data outliers. We repeat the previous experiment where we allow each approach to iteratively select 25 trials to benchmark. The machine is a production Bing Ads server, which runs intermittent background activities for software updates, monitoring, and periodic database updates. Resulting noises in the measurement can not reasonably modeled by normal distribution. Unlike comparison baselines, Metis proactively looks for potential data outliers after a sufficient amount of data points is collected, or 10 in our case.

Figure 7 illustrates that Metis has a higher chance of picking the optimal configuration in the presence of data outliers. The red line shows when Metis stops marking all sampled data points as potential data outliers (due to insufficient data collected). The effectiveness of Metis improves after the red line. The figure also shows the effectiveness of Metis without re-sampling, and removing outliers improves Metis's winning rate by 33.33%.

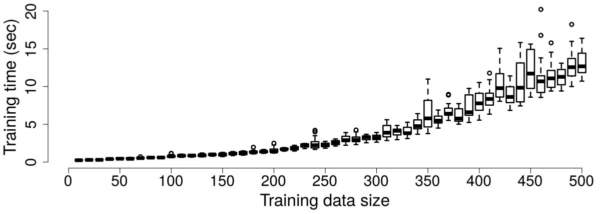


Figure 8: Impact of training data size on the training time of GP models.

6.3 Overhead of Metis

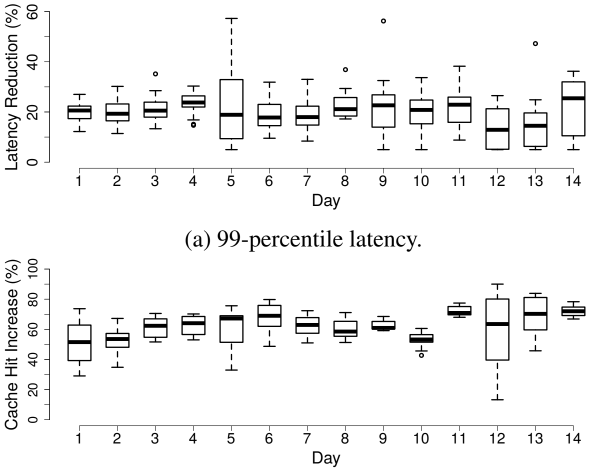
It is known that benchmarking configurations can be time-consuming due to system warm-up and stabilization. Another potential source of overhead is the predictive regression model – both in training and configuration selection. To quantify this overhead, we randomly generate the training data for individual experiment runs, and repeat each experiment 50 times.

Model training time. This is the time for Metis to fit a Gaussian process model over all sampled data points. We note that the community has shown that training the Gaussian process model can exhibit a complexity of $O(N^3 + N^2 D)$ [21], where N and D are the number and the dimensionality of training data points, respectively. This suggests that the number of training data points largely determines the training time. Our goal is to understand whether the training complexity will limit Metis's practicality in the real-world. Figure 8 shows that the training time increases with the number of sampled data points (for a training data set with a dimension of 20, and parameter values range between 1 and 1,000). When the number of sampled data points increases to 500, the training time reaches 12.33 seconds. However, this training time is still practical in real-world deployments.

Configuration selection time. This is the time for Metis to predict a sampling point that is expected to maximize the given acquisition function. It has been shown that the time complexity for prediction with the Gaussian process model is $O(N^2 + ND)$ [21], where N and D are the number and the dimensionality of training data points, respectively. When the training data size increases to 500, predicting the observation of an unsampled point takes $\sim$10.25 msec. This time suggests that the predictive model is feasible for real-world usages.

7 Case Study: BingKV

With a year of Metis operational experience, there are observations and lessons learned from adopting auto-tuning in Microsoft services. This section presents measurements and experiences from our running ex-



(a) 99-percentile latency.

(b) Cache hit rate.

Figure 9: Performance comparison between the previous LRU-based data store and the Metis-tuned BingKV, based on 14-day hourly performance logs.

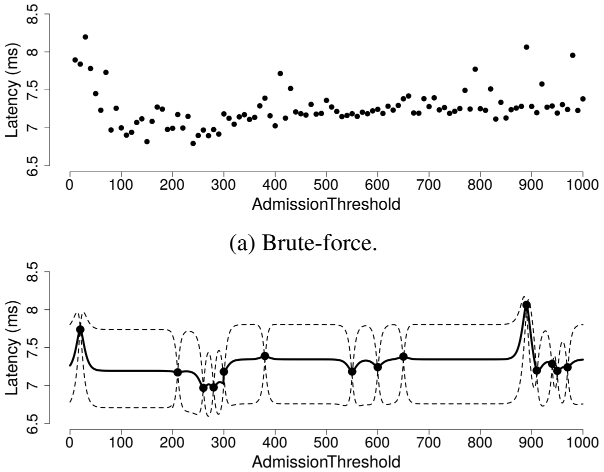
ample, Bing Ads KV store clusters (c.f. §2.1). The KV store evolved from LRU, expert-tuned BingKV, to Metis-tuned BingKV. We study measurements from two KV store clusters – $BingKV_1$ (∼700 servers handling ∼4.14 billion key queries per day) and $BingKV_2$ (∼1,700 servers handling ∼3.18 billion key queries per day). Each server of clusters has an in-memory cache capacity of 512 MB, and an SSD as the persistent data store.

Metis-tuned BingKV reduces the 99-percentile query lookup latency by an average of 20.4%, as compared to LRU. We start by comparing Metis-based KV store with the previously LRU-based KV store, under Bing Ads $BingKV_1$ production workload. The comparison is based on 14-day hourly logs of the 99-percentile query lookup latency and the cache hit rate (CHR). Figure 9 presents results for $BingKV_1$ – Figure 9a shows that Metis helps to reduce the 99-percentile latency by 20.4% on average (with a standard deviation of 8.4), and Figure 9b shows a CHR improvement of 60.6% (with a standard deviation of 11.7). This translates to a 22.76% reduction in disk I/O reads.

Metis-tuned BingKV reports a 3.43% lower 99-percentile latency, as compared to expert-tuned BingKV. Our human expert is one of the lead programmers for Bing Ads key-value store, with years of experience operating the system. As it is infeasible for the human expert to continuously tune the KV store, this subsection focuses on picking a single best-performing configuration. We note that the human expert did not have any time budget limitations, and manual tuning took about four weeks in a testing environment. The next subsection delves into the system tuning time comparison.

Perf metrics	**Differences**	
	$BingKV_1$	$BingKV_2$
99-percentile latency	-2.99%	-3.43%
Cache hit rate	2.43%	0.49%

Table 2: Performance comparison of expert-tuned and Metis-tuned BingKV-based data store, under 14-day Bing Ads workloads. Metis-tuned configurations outperform expert-tuned configurations, while reducing the tuning time from weeks to hours.



(a) Brute-force.

(b) Metis.

Figure 10: On average, Metis reduces the configuration tuning time by 98.41% for BingKV. For illustration, this figure shows a one-dimensional case of sampling points selected by Metis, as compared to brute-force.

Table 2 shows that Metis-tuning outperforms human-tuning under two weeks of production traffic: ***(1)*** the Metis-tuned configuration reports a 2.99% and 3.43% lower 99-percentile latency for $BingKV_1$ and $BingKV_2$, respectively. ***(2)*** Metis-tuned configuration reports a 2.43% and 0.49% higher CHR for $BingKV_1$ and $BingKV_2$, respectively.

Metis reduces the overall tuning time by 98.41%, as compared to manual tuning by human experts. While humans are typically guided by intuition based on their knowledge of the system, much of the manual tuning process mostly resembles the random search. In fact, the problem exacerbates as the dimensionality of the parameter space increases, especially in the case where parameters have dependencies. For reference, in one scenario deployment, manual-tuning took about 3 weeks, while Metis-tuning took about 8 hours (including sampling, modeling and prediction).

To illustrate how predictive modeling reduces the tuning time, Figure 10 shows that Gaussian process regression for a one-dimensional case. GP regression is able to estimate the impact of ShadowAdmissionSize without sampling the entire parameter space. We note that dotted lines in Figure 10b mark the 95% confidence interval for each point, which can guide both the space exploration and the stopping criteria. Finally, we note that GP regression operations are negligible as compared to system benchmarking – training typically takes ∼1.02 sec, and predicting the expected value of an unsampled data points takes ∼0.53 msec.

8 Related Work

Black-box parameter tuning services. Google Vizier [11] is an ongoing research project, and it supplies core capabilities to optimize hyper-parameters of machine learning models across Google. Similar to Google Vizier, SigOpt [18] is a startup that tunes hyper-parameters of ML models, but little technical details have been disclosed. Google Slicer [1] is a sharding service that dynamically generates the optimal resource assignment, but its goals are not parameter space exploration and sampling. Like Metis, BestConfig [23] uses stratified sampling. However, it heavily leans towards exploitation, as it assumes a high probability of finding better-performing configurations around the currently best-performing one.

In contrast, Metis focuses on providing a robust auto-tuning algorithm for cloud systems, and addresses challenges that systems introduce to the optimization problem. Metis has been used to tune parameters of Microsoft services and networking infrastructure.

Parameter tuning with Bayesian optimization. Metis builds on previous efforts that demonstrate the potential of applying Bayesian optimization and Gaussian process to auto-tuning. iTuned [10] uses Latin Hypercube Sampling (LHS) to sample the parameter space, and model the parameter space with Gaussian process models. OtterTune [2] uses a combination of supervised and unsupervised machine learning methods to reduce the parameter dimension, characterize observed workloads, and recommend configurations. CherryPick [3] follows a similar approach to BO and GP in selecting the best-performing cloud configuration for a given machine learning workload.

TPE [5] uses Bayesian optimization with Gaussian mixture model, instead of Gaussian process. It is suitable for cases where there are some dependencies among parameters. However, since TPE trains with a subset of the training data, it needs a larger amount of data to be collected for training effectively. Smart Hill-Climbing [22] improves Latin Hypercube Sampling, but the GP-based approach has been shown to outperform [10].

Metis introduces customizations to the framework of BO with GP. These include the diagnostic model to find potential data outliers for re-sampling, and a mixture of acquisition functions to balance exploitation, exploration and re-sampling.

9 Discussion

We now discuss limitations concerning the applicability of Metis to systems in general.

Support of different system parameter types. Some types of system parameters can be non-trivial to model with regression models – First, categorical parameters take on one of a fixed number of non-integer values such as boolean. Since categorical parameters are not continuous in nature, it can be difficult to model the relationship among possible values. While categorical parameters are out of scope for this paper, our current implementation conceptually treats each categorical value as a new target system. Second, some systems have multi-step parameters, where one single configuration requires the system to go through a specific sequence of value changes for one or more parameters. Metis does not currently support multi-step parameters.

Costs of changing system configurations. Applying configuration changes can incur costs for some systems. First, server reboots might be necessary after a configuration change, thus service interruptions. To handle this case, system administrators can decide to push a configuration change only if the new configuration is predicted to offer a certain level of performance improvement (e.g., 10% latency reduction). Administrators can also bound the cost of reconfiguration, e.g., by performing reconfigurations gradually over time, or by bounding the parameter space exploration by the distance from the current running configuration. Second, mis-predictions can result in system performance degradation. Fortunately, Gaussian Process offers two ways to gain insights regarding uncertainties – GP offers a confidence interval for each prediction, and a log-marginal likelihood score [15] to quantify the model fitness with respect to the training dataset.

10 Conclusion

This paper reports the design, implementation, and deployment of Metis, an auto-tuning service used by several Microsoft services for robust system tuning. We demonstrate the effectiveness of Metis with controlled experiments and real-world system workloads. Furthermore, real-world deployments show that Metis can significantly reduce the system tuning time.

References

[1] ADYA, A., MYERS, D., HOWELL, J., ELSON, J., MEEK, C., KHEMANI, V., FULGER, S., GU, P., BHUVANAGIRI, L., HUNTER, J., PEON, R., KAI, L., SHRAER, A., MERCHANT, A., AND LEV-ARI, K. Slicer: Auto-Sharding for Datacenter Applications. In *OSDI* (2016), USENIX.

[2] AKEN, D. V., PAVLO, A., GORDON, G. J., AND ZHANG, B. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD* (2017), ACM.

[3] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI* (2017), USENIX.

[4] AZIMI, J. Bayesian Optimization (BO).

[5] BERGSTRA, J., BARDENET, R., BENGIO, Y., AND KEGL, B. Algorithms for Hyper-Parameter Optimization. In *NIPS* (2011).

[6] CAFLISCH, R. E. Monte Carlo and Quasi-Monte Carlo Methods. In *Acta Numerica* (1988).

[7] CHU, L., CURSI, E. S. D., HAMI, A. E., AND EID, M. Reliability Based Optimization with Metaheuristic Algorithms and Latin Hypercube Sampling Based Surrogate Models. In *Applied and Computational Mathematics* (2015).

[8] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *SoCC* (2010), ACM.

[9] DEAN, J., AND BARROSO, L. A. The Tail at Scale. In *Communications of the ACM* (2013), ACM.

[10] DUAN, S., THUMMALA, V., AND BABU, S. Tuning Database Configuration Parameters with iTuned. In *VLDB* (2009), ACM.

[11] GOLOVIN, D., SOLNIK, B., MOITRA, S., KOCHANSKI, G., KARRO, J., AND SCULLEY, D. Google Vizier: A Service for Black-Box Optimization. In *KDD* (2017), ACM.

[12] MCKAY, M. D., BECKMAN, R. J., AND CONOVER, W. J. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. In *American Statistical Association and American Society for Quality* (2000).

[13] RASMUSSEN, C. E., AND WILLIAMS, C. K. I. *Gaussian Processes for Machine Learning*. the MIT Press.

[14] RYZHOV, I. O. On the Covergence Rates of Expected Improvement Methods. In *Operations Research* (2014).

[15] SCHIRRU, A., PAMPURI, S., NICOLAO, G. D., AND MCLOONE, S. Efficient Marginal Likelihood Computation for Gaussian Process Regression. In *arXiv:1110.6546* (2011).

[16] SCIKIT-LEARN. GaussianProcessRegressor. `http://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.GaussianProcessRegressor.html`.

[17] SHAHRIARI, B., SWERSKY, K., WANG, Z., ADAMS, R. P., AND DE FREITAS, N. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proceedings of the IEEE* (2016).

[18] SIGOPT. SigOpt. `http://sigopt.com/`.

[19] SNOEK, J., LAROCHELLE, H., AND ADAMS, R. P. Practical Bayesian Optimization of Machine Learning Algorithms. In *NIPS* (2012).

[20] VENKATARAMAN, S., YANG, Z., FRANKLIN, M., RECHT, B., AND STOICA, I. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytic. In *NSDI* (2016), USENIX.

[21] WILSON, A. G., HU, Z., SALAKHUTDINOV, R., AND XING, E. P. Deep Kernel Learning. In *AISTATS* (2016).

[22] XI, B., LIU, Z., RAGHAVACHARI, M., XIA, C. H., AND ZHANG, L. A Smart Hill-Climbing Algorithm for Application Server Configuration. In *WWW* (2004).

[23] ZHU, Y., LIU, J., GUO, M., BAO, Y., MA, W., LIU, Z., SONG, K., AND YANG, Y. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *SoCC* (2017), ACM.

Redesigning LSMs for Nonvolatile Memory with NoveLSM

Sudarsun Kannan
University of Wisconsin-Madison

Nitish Bhat
Georgia Tech

Ada Gavrilovska
Georgia Tech

Andrea Arpaci-Dusseau
University of Wisconsin-Madison

Remzi Arpaci-Dusseau
University of Wisconsin-Madison

Abstract

We present NoveLSM, a persistent LSM-based key-value storage system designed to exploit non-volatile memories and deliver low latency and high throughput to applications. We utilize three key techniques – a byte-addressable skip list, direct mutability of persistent state, and opportunistic read parallelism – to deliver high performance across a range of workload scenarios. Our analysis with popular benchmarks and real-world workload reveal up to a 3.8x and 2x reduction in write and read access latency compared to LevelDB. Storing all the data in a persistent skip list and avoiding block I/O provides more than 5x and 1.9x higher write throughput over LevelDB and RocksDB. Recovery time improves substantially with NoveLSM's persistent skip list.

1 Introduction

Persistent key-value stores based on log-structured merged trees (LSM), such as BigTable [13], LevelDB [4], HBase [2], Cassandra [1], and RocksDB [3], play a crucial role in modern systems for applications ranging from web-indexing, e-commerce, social networks, down to mobile applications. LSMs achieve high throughput by providing in-memory data accesses, buffering and batching writes to disk, and enforcing sequential disk access. These techniques improve LSM's I/O throughput but are accompanied with additional storage and software-level overheads related to logging and compaction costs. While logging updates to disk before writing them to memory is necessary to recover from application or power failure, compaction is required to restrict LSM's DRAM buffer size and importantly commit non-persistent in-memory buffer to storage. Both logging and compaction add software overheads in the critical path and contribute to LSM's read and write latency. Recent proposals have mostly focused on redesigning LSMs for SSD to improve throughput [23, 30, 40].

Adding byte-addressable, persistent, and fast non-volatile memory (NVM) technologies such as PCM in the storage stack creates opportunities to improve latency, throughput, and reduce failure-recovery cost. NVMs are expected to have near-DRAM read latency, 50x-100x faster writes, and 5x higher bandwidth compared to SSDs. These device technology improvements shift performance bottlenecks from the hardware to the software stack, making it critical to reduce and eliminate software overheads from the critical path of device accesses. When contrasting NVMs to current storage technologies, such as flash memory and hard-disks, NVMs exhibit the following properties which are not leveraged in current LSM designs: (1) random access to persistent storage can deliver high performance; (2) in-place update is low cost; and (3) the combination of low-latency and high bandwidth leads to new opportunities for improving application-level parallelism.

Given the characteristics of these new technologies, one might consider designing a new data structure from scratch to optimally exploit the device characteristics. However, we believe it worthwhile to explore how to redesign LSMs to work well with NVM for the following reasons. First, NVMs are expected to co-exist with large-capacity SSDs for the next few years [27] similar to the co-existence of SSDs and hard disks. Hence, it is important to redesign LSMs for heterogeneous storage in ways that can exploit the benefits of NVMs without losing SSD and hard disk optimizations. Second, redesigning LSMs provides backward compatibility to thousands of applications. Third, maintaining the benefits of batched, sequential writes is important even for NVMs, given the 5x-10x higher-than-DRAM write latency. Hence in this paper, we redesign existing LSM implementations.

Our redesign of LSM technology for NVM focuses on the following three critical problems. First, existing LSMs maintain different in-memory and persistent storage form of the data. As a result, moving data across storage and memory incurs significant serialization and deserialization cost, limiting the benefits of low latency NVM. Second, LSMs and other modern applications [1–4, 13] only allow changes to in-memory data structures and make the data in persistent storage immutable. However, memory buffers are limited in their capacity and must be frequently compacted, which increases stall time. Buffering data in memory can result in loss of data after a system failure, and hence updates must be logged; this increases latency, and leads to I/O read and write amplification. Finally, adding NVM to the LSM hierarchy increases the number of levels in the storage hierarchy which can increase read-access latency.

To address these limitations, we design **NoveLSM**, a persistent LSM-based key-value store that exploits the byte-addressability of NVMs to reduce read and write latency and consequently achieve higher throughput. NoveLSM achieves these performance gains through three key innovations. First, NoveLSM introduces a *persistent NVM-based memtable*, significantly reducing the serialization and deserialization costs which plague standard LSM designs. Second, NoveLSM makes the *persistent NVM memtables mutable*, thus allowing direct updates; this significantly reduces application stalls due to compaction. Further, direct updates to NVM memtable are committed in-place, avoiding the need to log updates; as a result, recovery after a failure only involves mapping back the persistent NVM memtable, making it three orders of magnitude faster than LevelDB. Third, NoveLSM introduces *optimistic parallel read*s to simultaneously access multiple levels of the LSM that can exist in NVM or SSD, thus reducing the latency of read requests and improving the throughput of the system.

We build NoveLSM by redesigning LevelDB, a widely-used LSM-based key-value store [4]. NoveLSM's design principles can be easily extended to other LSM implementations [1–3]. Our analysis reveals that NoveLSM significantly outperforms traditional LSMs when running on an emulated NVM device. Evaluation of NoveLSM with the popular DBbench [3,4] shows up to 3.8x improvement in write and up to 2x improvement in read latency compared to a vanilla LevelDB running on an NVM. Against state-of-the-art RocksDB, NoveLSM reduces write latency by up to 36%. When storing all the data in a persistent skip list and avoiding block I/O to SSTable, NoveLSM provides more than 5x and 1.9x gains over LevelDB and RocksDB. For the real-world YCSB workload, NoveLSM shows a maximum of 54% throughput gain for scan workload and an average of 15.6% across all workloads over RocksDB. Finally, the recovery time after a failure reduces significantly.

2 Background

We next provide background on LSM trees and on the design of popular LSM stores, LevelDB [4] and RocksDB [3], used extensively in this work. We also present a background on persistent memory and our method of emulating it.

2.1 Log Structured Merge Trees

An LSM-tree proposed by O'Neil et al. [32] is a persistent structure that provides efficient indexing for a key-value store. LSMs achieve higher write throughput by first staging data in memory and then across multiple levels on disk to avoid random disk writes. In LSMs, the levels have an increasing size; for example, in LevelDB, each level is at least ten times larger than the previous level. During an insert operation, the keys are first inserted into an in-memory level, and as this level fills up, the data slowly trickles down to disk-friendly block structures of the lower levels, where data is always sorted. Before every insert operation into the memory level, the data (key-value pairs) is logged in the persistent storage for recovery after a failure; the logs are garbage collected after data is safely flushed and persisted to on-disk structures. Next, the search and read operations proceed from the top memory level to the disk levels and their latency increases with increasing number of levels. In general, LSMs are update-friendly data structures and read operations are comparatively slower to other NoSQL designs.

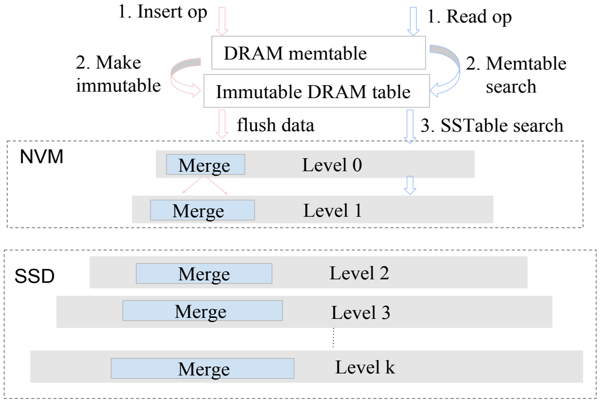


Figure 1: **Naive LevelDB design with NVM.** *Figure shows a simple method to add NVM to the LSM hierarchy. NVM is used only as a replacement to disk for storing SSTables. Shaded blocks show immutable storage, grey and red arrows show steps for read operation search and the background compaction.*

2.2 Popular LSM Stores

LevelDB is a popular LSM-based key-value store derived from Google's BigTable implementation and is widely-used from browsers to datacenter applications. Figure 1 shows LevelDB's design with NVM added to the storage hierarchy. During an insert operation, LevelDB buffers updates in a memory-based skip list table (referred to as memtable hereafter) and stores data on multiple levels of on-disk block structures know as sorted string tables (SSTable). After the memtable is full, it is made immutable and a background compaction thread moves the immutable memtable data to on-disk SSTable by serializing the data to disk-based blocks. Only two levels of memory tables (mutable and immutable) exist. With an exception to memtables, all lower levels are mutually exclusive and do not maintain redundant data.

The SSTables are traditional files with a sequence of I/O blocks that provide a persistent and ordered immutable mapping from keys to values, as well as interfaces for sequential, random, and range lookup operations. The SSTable file also has a block index for locating a block in $O(1)$ time. A key lookup can be performed with a single disk seek to read the block and binary search inside the block. In LevelDB, for read operations, the files with SSTables are memory-mapped to reduce the POSIX file system block-access overheads.

RocksDB is an LSM implementation that extends LevelDB to exploit SSD's high bandwidth and multicore parallelism. RocksDB achieves this by supporting multithreaded background compaction which can simultaneously compact data across multiple levels of LSM hierarchy and extracts parallelism with multi-channel SSDs. RocksDB is highly configurable, with additional features such as compaction filters, transaction logs, and incremental replication. The most important feature of RocksDB that can be beneficial for NVM is the use of a Cuckoo hashing-based SST table format optimized for random lookup instead of the traditional I/O block-based format with high random-access overheads.

In this work, we develop NoveLSM by extending LevelDB. We choose LevelDB due to its simplicity as well as its broad usage in commercial deployments. The optimizations in RocksDB over LevelDB are complementary to the proposed NoveLSM design principles.

2.3 Byte-addressable NVMs

NVM technologies such as PCM are byte-addressable persistent devices expected to provide 100x lower read and write latency and up to 5x-10x higher bandwidth compared to SSDs [7, 12, 17, 22]. Further, NVMs can scale to 2x-4x higher density than DRAM [5]. These attributes make NVM a suitable candidate for replacing SSDs. Additionally, NVMs are expected to be placed in parallel with DRAM connected via the memory bus, thereby providing memory-like load/store access interface that can avoid POSIX-based block access supported in current storage devices. Further, the read (load) latency of NVMs is comparable to DRAM, but the write latency is expected be to 5x slower.

2.4 NVM Emulation

Since byte-addressable NVMs are not available commercially, we emulate NVMs similarly to prior research [12, 17, 22, 26] and our emulation methodology uses a 2.8 GHz, 32-core Intel Nehalem platform with 25 MB LLC, dual NUMA socket with each socket containing 16 GB DDR3 memory, and an Intel-510 Series SSD. We use Linux 4.13 kernel running DAX-enabled Ext4 [6] file system designed for persistent memory. We use one of the NUMA sockets as NVM node, and to emulate lower NVM bandwidth compared to DRAM, we thermal throttle the NUMA socket [25]. To emulate higher write latency, we use a modified version of NVM emulator [37] and inject delay by estimating the number of processor store cache misses [12, 17, 22]. For our experiments, we emulate 5x higher NVM write latency compared to DRAM access latency and keep the NVM read latency same as the DRAM latency. We vary NVM bandwidth from 2 GB/s to 8 GB/s; the 8 GB/s bandwidth is same as DRAM's per-channel bandwidth and is considered an ideal case.

3 Motivation

NVMs are expected to provide an order of magnitude lower latency and up to 8x higher bandwidth compared to SSDs; but can the current LSM software stack fully exploit the hardware performance benefits of NVM? To understand the impact of using NVM in current LSM designs, we analyze LevelDB's performance by using NVM for its persistent storage. We use the widely-used DBbench [4, 30, 35] benchmark with the total key-value database size set to 16 GB, and the value size set to 4 KB. Figure 2 compares the latency of sequential and random LSM write and read operations. We configure the maximum size of each SSTable file to 64 MB, a feature recently added to LevelDB to improve read performance [4].

As shown in Figure 2, although NVM hardware provides 100x faster read and write compared to SSD, LevelDB's sequential and random insert latency (for 5 GB/sec bandwidth) reduce by just 7x and 4x, respectively; the sequential and random read (fetch) latency reduces by less than 50%. The results show that *current LSMs do not fully exploit the hardware benefits of NVM and suffer from significant software overheads*. We next decipher the sources of these overheads.

Insert latency. A key-value pair insert (or update) operation to LSM is first buffered in the memory – mutable memtable (skip list in LevelDB) – before writing the key-value pair to the storage layer (SSTables). However, a power failure or a system crash can lead to data loss (buffered in memory). To avoid data loss, the key-value pairs and their checksum are first added to a sequential log in the persistent storage before inserting them to the memtable. When the memtable is full, it is made immutable, and a new mutable memtable is created to which new inserts continue. A background thread compacts the immutable memtable to storage; however, if the new mutable memtable also fills up before the completion of background compaction, all new inserts to LSM are stalled. Current LSM designs suffer from high compaction cost because compaction involves iterating the immutable memtable skip list, serializing data to disk-compatible (SSTable) format, and

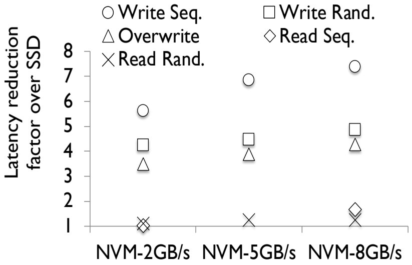


Figure 2: **Latency reduction factor.** *Analysis shows LevelDB using NVM for storage compared to SSD for 4 KB values; x-axis varies NVM bandwidth.*

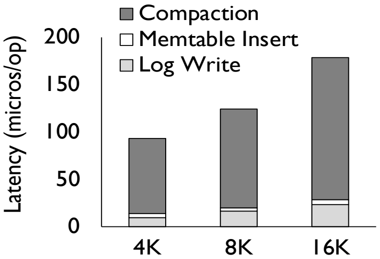


Figure 3: **Write latency cost split-up.** *Compaction happens in the background but stalls LSM when memtables are full and compaction is not complete.*

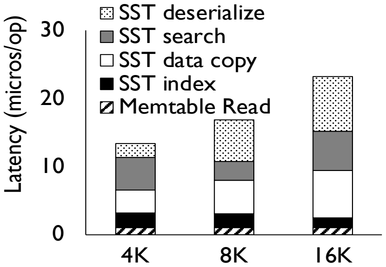


Figure 4: **Read latency cost split-up.** *SST denotes SSTable, and the graph shows time spent on memtable and different SSTable methods.*

finally committing them to the storage. Besides, the storage layer (SSTable) comprises of multiple levels, and the memtable compaction can trigger a chain of compaction across these levels, stalling all foreground updates.

Figure 3 shows the cost breakup for insert operations with 4 KB, 8 KB, and 16 KB values. As shown in the figure, data compaction dominates the cost, increasing latency by up to 83%, whereas log writes and checksum calculations add up to 17% of the total insert latency. Increasing the in-memory buffer (memtable) can reduce compaction frequency; however, this introduces several drawbacks. First, DRAM usage increases by two times: memory must be increased for both mutable and immutable memtables. Second, only after the immutable memtable is compacted, log updates can be cleaned, leading to a larger log size. Third, LSMs such as LevelDB and RocksDB do not enforce commits (sync) when writing to a log; as a result, an application crash or power-failure could lead to data loss. Fourth, a larger log also increases recovery time after a failure. Finally, the cost of checksumming and logging also increases.

Read operation latency. A read operation involves hierarchically searching the smaller in-memory mutable and immutable memtables, followed by searching multiple SSTable levels. Searching a memtable involves traversing the skip list without the need to deserialize data. However, searching a SSTable is complicated for the following reason: the SSTable contains multiple levels that store key-value pairs sorted by their key hash, and each level is searched using a binary search. After locating the blocks containing a key-value pair, the blocks are copied into a memory buffer and then deserialized from disk to an in-memory format. The search cost increases moving top-down across SSTable levels because each SSTable level is at least 10x larger than the previous level. To reduce SSTable search cost, LSMs such as LevelDB and RocksDB maintain an index table at each level which uses a Bloom filter to cache recently searched keys, which is useful only for workloads with high re-access rates (e.g., Zipfian distribution). Figure 4 breaks down the cost of a read operation for 4 KB, 8 KB, and 16 KB values. For small values, searching the SSTable dominates the cost, followed by copying disk blocks to memory and deserializing block contents to in-memory key-value pairs; the deserialization cost increases with increasing value size (e.g., 16 KB). Reducing data copy, search, and deserialization cost can significantly reduce read latencies.

Summary. To summarize, existing LSMs suffer from high software overheads for both insert and read operations and fail to exploit NVM's byte-addressability, low latency, and high storage bandwidth. The insert operations suffer mainly from high compaction and log update overheads, and the read operations suffer from sequential search and deserialization overheads. Reducing these software overheads is critical for fully exploiting the hardware benefits of NVMs.

4 Design

Based on the analyses presented earlier, we first formulate NoveLSM's design principles and then discuss the details on how these principles are incorporated to NoveLSM's design.

4.1 NoveLSM Design Principles

NoveLSM exploits NVMs byte addressability, persistence, and large capacity to reduce serialization and deserialization overheads, high compaction cost, and logging overheads. Further, NoveLSM utilizes NVM's low latency and high bandwidth to parallelize search operations and reduce response time.

Principle 1: Exploit byte-addressability to reduce serialization and deserialization cost. NVMs provide byte-addressable persistence; therefore, in-memory structures can be stored in NVM as-is without the need to serialize them to disk-compatible format or deserialize them to memory format during retrieval. To exploit this, NoveLSM provides a persistent NVM memtable by designing a persistent skip list. During compaction, the DRAM memtable data can be directly moved to the NVM memtable without requiring serialization or deserialization.

Principle 2: Enable mutability of persistent state and leverage large capacity of NVM to reduce compaction cost. Traditionally, software designs treat data in the storage as immutable due to high storage access latency; as a result, to update data in the storage, data must be read into a memory buffer before making changes and writing them back (mostly in batches). However, NVM byte-addressability provides an opportunity to directly update data on the storage without the need to read them to a memory buffer or write them in batches. To exploit mutability of persistent state, NoveLSM designs a large mutable persistent memtable to which applications can directly add or update new key-value pairs. The persistent memtable allows NoveLSM to alternate between small DRAM and large NVM memtable without stalling for background compaction to complete. As a result compaction cost significantly reduces.

Principle 3: Reduce logging overheads and recovery cost with in-place durability. Current LSM designs must first write updates to a log, compute the checksum and append them, before inserting them into the memtable. Most LSMs compromise crash consistency for performance by not committing the log updates. Further, recovery after an application failure or system crash is expensive; each log entry must be deserialized before adding it to the memtable. In contrast, NoveLSM avoids logging by immediately committing updates to the persistent memtable in-place. Recovery is fast and only requires memory mapping the entire NVM memtable without deserialization.

Principle 4: Exploit the low latency and high bandwidth of NVM to parallelize data read operations. LSM stores data in a hierarchy with top in-memory levels containing new updates, and older updates in the lower SSTables levels. With an increase in the number of key-value pairs in a database, the number of storage levels (i.e., SSTables) increases. Adding NVM memtables further increases the number of LSM levels. LSMs must be sequentially searched from top to bottom, which can add significant search costs. NoveLSM exploits NVMs' low latency and high bandwidth by parallelizing search across the memory and storage levels, without affecting the correctness of read operations.

4.2 Addressing (De)serialization Cost

To reduce serialization and deserialization cost in LSMs, we first introduce an immutable persistent memtable. During compaction, each key-value pair from the DRAM memtable is moved (via *memcpy()*) to the NVM memtable without serialization. The NVM memtable skip list nodes (that store key-value pairs) are linked by their relative offsets in a memory-mapped region instead of virtual address pointers and are committed in-place; as a result, the persistent NVM skip list can be safely recovered and rebuilt after a system failure. Figure 5.a shows the high-level design of an LSM with NVM memtable placed behind DRAM memtable.

Immutable NVM skip list-based memtable. We design a persistent memtable by extending LevelDB's skip list and adding persistence support. A skip list is a multi-dimensional linked-list that provides fast probabilistic insert and search operation avoiding the need to visit all elements of a linked list [33]. Popular LSM implementations, such as LevelDB and RocksDB, use a skip list because they perform consistently well across sequential, random, and scan workloads. We extend the skip list for persistence because it enables us to reuse LevelDB's skip list-specific optimizations such as aligned memory allocation and faster range queries.

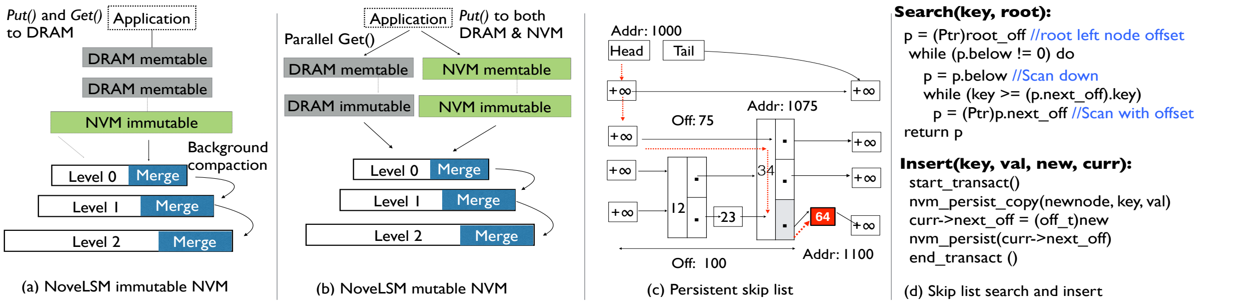
In a persistent skip list, the nodes are allocated from a large contiguous memory-mapped region in the NVM. As shown in Figure 5.d, each skip list node points to a next node using physical offset relative to the starting address of the root node, instead of a virtual address. Iterating the persistent skip list requires root node's offset from starting address of the memory-mapped region. After a restart or during failure recovery, the persistent region is remapped, and the root offset is recovered from a log file; using the root node, all skip list nodes are recovered.

To implement a persistent skip list, we modify LevelDB's memtable with a custom persistent memory NVM allocator that internally uses the Hoard allocator [10]. Our allocator internally maps a large region of NVM pages on a DAX filesystem [6] and manages the pages using persistent metadata similar to Intel's NVML library [24]. Each skip list node maintains a physical offset pointer and a virtual address pointer to the next node, which are updated inside a transaction during an insert or update operation, as shown in Figure 5.d. A power or application failure in the middle of a key-value pair insertion or the offset update can compromise durability. To address this, we provide ordered persistent updates by using hardware memory barriers and cacheline flush instructions [15, 16, 22, 38]. Note that NoveLSM extends existing LSMs for NVMs rather than completely redesigning their data structures; this is complementary to prior work that focuses on optimizing LSMs' in-memory data structures [9, 34].

4.3 Reducing Compaction Cost

Although the immutable NVM design can reduce serialization cost and read latency, it suffers from several limitations. First, the NVM memtable is just a replica of the DRAM memtable. Hence, the compaction frequency is dependent on how fast the DRAM memtables fill. For applications with high insert rates, compaction cost dominates the performance.

Mutability for persistent memtable. To address the issue of compaction stalls, NoveLSM makes the NVM memtable mutable, thereby allowing direct updates to the NVM memtable (Figure 5.(b)); when the in-memory



(a) NoveLSM immutable NVM

(b) NoveLSM mutable NVM

(c) Persistent skip list

```
Search(key, root):
  p = (Ptr)root_off //root left node offset
   while (p.below != 0) do
     p = p.below //Scan down
     while (key >= (p.next_off).key)
       p = (Ptr)p.next_off //Scan with offset
  return p

Insert(key, val, new, curr):
   start_transact()
   nvm_persist_copy(newnode, key, val)
   curr->next_off = (off_t)new
   nvm_persist(curr->next_off)
   end_transact ()
```

(d) Skip list search and insert

Figure 5: **NoveLSM's immutable and mutable NVM memtable design.** *(a) shows the immutable memtable design, where NVM memtable is placed below DRAM memtable to only store compaction data. (b) shows mutable memtable design where inserts can be directly updated to NVMs persistent skip list. (c) shows an example of inserting a key 64 to persistent NVM memtable at offset 100 from the root node with address 1000. Next pointer of node with key 34 contains offset 100. (d) shows NVM persistent skiplist search and insert pseudocode. Read operation searches for a key using the offset of node instead of pointer. Offsets represent the location of a node in a memory-mapped file. Inserts to NVM memtable are committed in-place and avoid separate logging.*

memtable is full, application threads can alternate to using the NVM memtable without stalling for the in-memory memtable compaction to complete.

The working of the mutable NVM design can be summarized as follows. During initialization, NoveLSM creates a volatile DRAM memtable and a mutable persistent NVM memtable. Current LSM implementations use a smaller memtable size to reduce DRAM consumption and avoid data loss after a failure. In contrast, NoveLSM uses a large NVM memtable; this is because NVMs can scale up to 4x larger than DRAM and also maintain persistence. To insert a key-value pair, first, the DRAM memtable is made active; the key-value pairs and their checksum are written to a log and then inserted into the DRAM memtable. When the DRAM memtable is full, it is made immutable, and the background compaction thread is notified to move data to the SSTable. Instead of stalling for compaction to complete, NoveLSM makes NVM memtable active (mutable) and keys are directly added to mutable NVM memtable. The large capacity of NVM memtable provides sufficient time for background compaction of DRAM and NVM immutable memtables without stalling foreground operations; as a result, NoveLSM's mutable memtable design significantly reduces compaction cost leading to lower insert/update latency. For read operations, the most recent value for a key is fetched by first searching the current active memtable, followed by immutable memtables and SSTables.

4.4 Reducing Logging Cost

NoveLSM eliminates logging for inserts added to mutable NVM memtable by persisting updates in-place. As a result, NoveLSM reduces the number of writes for each key-value pair and also reduces recovery time. We next discuss the details.

Logging. In current LSM implementations, each key-value pair and its 32-bit CRC checksum is first appended to a persistent log, then inserted into the DRAM memtable, and finally compacted to an SSTable, leading to high write amplification. Further, popular implementations such as LevelDB and RocksDB only append but do not commit (*fsync()*) data to the log; as a result, they compromise durability for better performance.

In contrast, for NoveLSM, when inserting into the mutable persistent memtable in NVM, all updates are written and committed in-place without requiring a separate log; as a result, NoveLSM can reduce write amplification. Log writes are avoided only for updates to NVM memtable, whereas, all inserts to the DRAM memtable are logged. Our evaluations show that using a large NVM memtable with direct mutability reduces logging to a small fraction of the overall writes, thereby significantly reducing logging cost and recovery time. Additionally, because all NVM updates are committed in-place, NoveLSM can provide stronger durability guarantees compared to existing implementations. Figure 5.d shows the pseudocode for NVM memtable insert. First, a new transaction is initiated and persistent memory for a key-value pair is allocated. The key-value pair is copied persistently by ordering the writes using memory store barrier and cache line flush of the destination addresses, followed by a memory store barrier [24, 38, 41]. As an additional optimization, small updates to NVM (8-byte) are committed with atomic store instruction not requiring a barrier. Finally, for overwrites, the old nodes are marked for lazy garbage collection.

Recovery. Recovering from a persistent NVM memtable requires first mapping the NVM memtable (a file) and identifying the root pointer of the skip list. Therefore, the NVM memtable root pointer offset and 20-bytes of skip list-related metadata are stored in a separate file. With a volatile DRAM and persistent NVM memtable, a failure can occur while a key with version V_i in the persistent NVM memtable is getting overwritten in the DRAM memtable to version V_{i+1}. A failure can also occur while a key with version V_i in the DRAM memtable, not yet compacted to storage, is getting overwritten to version V_{i+1} in the NVM. To maintain correctness, NoveLSM must always recover from the greatest committed version of the key. To achieve version correctness, NoveLSM performs the following steps: (1) a new log file is created every time a new DRAM memtable is allocated and all updates to DRAM memtable are logged and made persistent; (2) when the NVM memtable (which is a persistent skip list on a memory-mapped file) is made active, inserts are not logged, and the NVM memtable is treated as a log file; (3) the NVM log files are also named with an incremental version number similar to any other log file. During LSM restart or failure recovery, NoveLSM starts recovering from the active versions of log files in ascending order. A key present in a log file log_{i+1}, which could be either a DRAM log or NVM memtable, is considered as the latest version of the key. Note that recovering data from NVM memtable only involves memory-mapping the NVM region (a file) and locating the skip list root pointer. Therefore, recovering from even a large NVM memtable is fast with almost negligible cost of mapping pages to the page tables.

4.5 Supporting Read Parallelism

NoveLSM leverages NVM low latency and high bandwidth to reduce the latency of each read operation by parallelizing search across memtables and SSTables. In this pursuit, NoveLSM does not compromise the correctness of read operation. In current LSMs, read operations progress top-down from memtable to SSTables. A read miss at each level increases read latency. Other factors such as deserializing data from the SSTable also add to read overheads.

Reducing read latency. To reduce read latency, NoveLSM takes inspiration from the processor design, which parallelizes cache and TLB lookup to reduce memory access latency for cache misses; NoveLSM parallelizes search across multiple levels of LSM: DRAM and NVM mutable memtables, DRAM and NVM immutable tables, and SSTables. Our design manages a pool of worker threads that search memtables or the SSTable. Importantly, NoveLSM uses only one worker thread for searching across the mutable DRAM and NVM memtable because of the relatively smaller DRAM memtable size compared to the NVM memtable.

With this design, the read latency is reduced from $T_{read} \approx T_{mem_{DRAM}} + T_{mem_{NVM}} + T_{imm} + T_{SST}$ to $T_{read\_parallel} \approx max(T_{mem_{DRAM}} + T_{mem_{NVM}}, T_{imm}, T_{SST}) + C$. $T_{mem_{DRAM}}$, $T_{mem_{NVM}}$, T_{imm}, and T_{SST} represent the read time to search across the DRAM and NVM mutable memtable, the immutable memtable, and the SSTable, and C represents a constant corresponding to the time to stop other worker threads once a key has been found.

Guaranteeing version correctness for reads. Multiple versions of a key can exist across different LSM levels, with a newer version (V_{i+1}) of the key at the top LSM level (DRAM or NVM mutable memtable) and older versions ($V_i, V_{i-1}, ...$) in the lower immutable memtable and SSTables. In traditional designs, search operations sequentially move from the top memtable to lower SSTables, and therefore, always return the most recent version of a key. In NoveLSM, search operations are parallelized across different levels and a thread searching the lower level can return with an older version of the key; this impacts the correctness of read operation. To guarantee version correctness, NoveLSM always considers the value of a key returned by a thread accessing the highest level of LSM as the correct version. To satisfy this constraint, a worker thread T_i accessing L_i is made to wait for other worker threads accessing higher levels L_0 to L_{i-1} to finish searching, and only if higher levels do not contain the key, the value fetched by T_i is returned. Additionally, stalling higher-level threads to wait for the lower-level threads to complete can defeat the benefits of parallelizing read operation. To overcome this problem, in NoveLSM, once a thread succeeds in locating a key, all lower-level threads are immediately suspended.

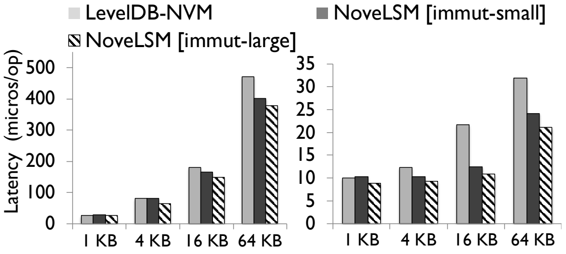
Optimistic parallelism and management. Introducing parallelism for each read operation is only beneficial when the overheads related to thread management cost are significantly lower than the actual cost to search and read a key-value pair. NoveLSM uses an optimistic parallelism technique to reduce read latency.

Thread management cost. In NoveLSM, the main LSM thread adds a client's read request to a job pool, notifies all worker threads to service the request, and finally, returns the value for a key. NoveLSM always colocates the master and the worker threads to the same CPU socket to avoid the lock variable bouncing across processor caches on different sockets. Further, threads dedicated to parallelize read operation are bound to separate CPUs from threads performing backing compaction. These simple techniques are highly effective in reducing thread management cost.

Optimistic parallelism. While the thread pool optimizations reduce overheads, using multiple threads for keys that are present in DRAM or NVM memtable only adds more overheads. To avoid these overheads, we implement a Bloom filter for NVM and DRAM memtable. The Bloom filter predicts likeliness of a key in the memtable, and read parallelism is enabled only when a key is predicted to miss the DRAM or NVM memtable; false positives (keys that are predicted to be in memtable but are not present) only make the read operations sequential without compromising correctness.

5 Evaluation

Our evaluation of NoveLSM aims to demonstrate the design insights in reducing write and read latency and increasing throughput when using NVMs. We answer the following important questions.



(a) Write latency (b) Read latency

Figure 6: **Write and Read latency.** *Impact of NVM immutable for random writes and reads. Database size is constant.*

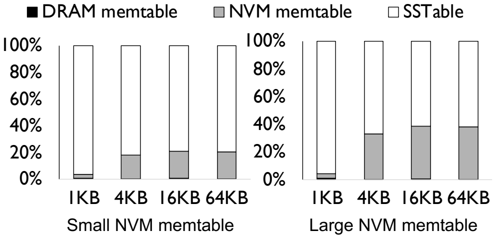


Figure 7: **NoveLSM immutable memtable hits.** *Figure shows percentage split of keys read from different LSM levels when using the immutable memtable design.*

1. What are the benefits of introducing a persistent immutable NVM memtable for different access patterns?
2. Does enabling mutability for NVM memtable reduce compaction cost and improve performance?
3. How effective is NoveLSM's optimistic parallelism in reducing read latency?
4. What is the impact of splitting NoveLSM across NVM and SSD compared to state-of-the-art approaches?
5. Is NoveLSM effective in exploiting NVMs byte-addressability to make failure recovery faster?

We first describe our evaluation methodology, and then evaluate NoveLSM with benchmarks and realistic workloads.

5.1 Methodology and Workloads

For our evaluation, we use the same platform described earlier § 2.4. NoveLSM reserves and uses 16 GB (a memory socket) to emulate NVM with 5 GB/sec NVM bandwidth and the read and write latency set to 100ns and 500ns respectively, similarly to [12, 17, 22], using methodology described earlier. We evaluate NoveLSM using DBbench [3, 4, 30] and the YCSB cloud benchmark [18]. The total LSM database size is restricted to 16 GB to fit in the NUMA socket that emulates NVM. The key size (for all key-values) is set to 16 bytes and only the value size is varied. We turn off database compression to avoid any undue impact on the results, as done previously [30].

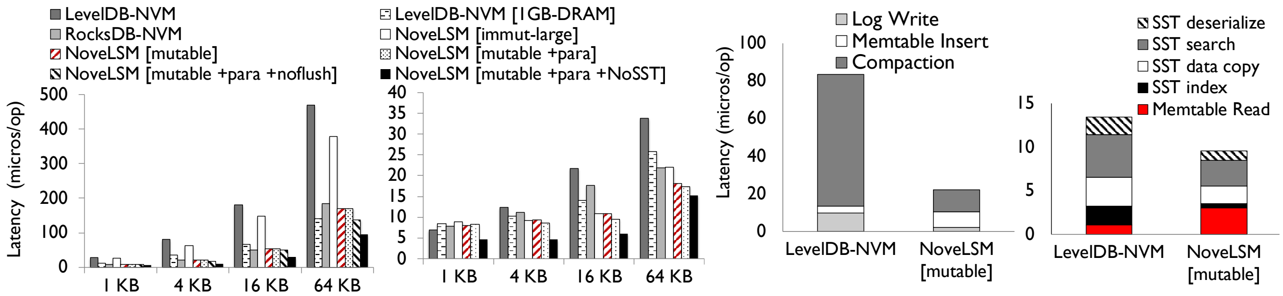
5.2 Impact of NVM-immutable Memtable

We begin by evaluating the benefits and implications of adding a persistent NVM immutable to the LSM hierarchy. We study two versions of NoveLSM: NoveLSM with a small (2 GB) immutable NVM memtable (NoveLSM+immut-small), and NoveLSM with a large (4 GB) immutable NVM memtable (NoveLSM+immut-large). The remaining NVM space is used for storing SSTables. For comparison, we use a vanilla LevelDB that stores all its non-persistent data in a DRAM memtable and persistent SSTables in the NVM (LevelDB-NVM). Figures 6.a and 6.b show the average random write and read latency as a function of the value sizes in X-axis.

Random write latency. Figure 6.a compares the random write latency. For the naive LevelDB-NVM, when the in-memory (DRAM) immutable memtable is full, a compaction thread first serializes data to SSTable. In contrast, NoveLSM uses a persistent NVM immutable memtable (a level below the 64 MB DRAM immutable memtable). When the DRAM immutable memtable is full, first data is inserted and flushed to NVM memtable skip list without requiring any serialization. When NVM memtable is also full, its contents are serialized and flushed to SSTable by a background thread. Using a larger NVM memtable (NoveLSM+immut-large) as a buffer reduces the memory to disk format compaction cost but without compromising crash consistency. Therefore, the NVM immutable design achieves up to 24% reduction in latency for 64 KB value compared to LevelDB-NVM. However, due to lack of direct NVM memtable mutability, the compaction frequency is dependent on the DRAM memtable capacity, which impacts immutable NVM designs performance.

Random read latency. Figure 6.b shows the read latency results. In case of LevelDB-NVM, reading a key-value pair from SSTable requires first locating the SSTable level, searching for the key within a level, reading the corresponding I/O blocks, and finally deserializing disk blocks to in-memory data. NoveLSM's immutable memtable skip list also incurs search cost; however, it avoids indexing, disk block read, and deserialization cost. Figure 7 shows the NVM immutable table hit rate for different value sizes when using small and large NVM tables. For 4 KB value size, the memtable hit rate (DRAM or NVM) for small NVM memtable is less than 17% and the additional search in the NVM memtable increases latency. However, for NoveLSM+immut-large, the hit rate is around 29% and the read latency reduces by 33% compared to LevelDB-NVM. Because we keep the database size constant and vary the value size, for larger value sizes (e.g., 64 KB), the number of key-values in the database is less, increasing hit rate by up to 38% and reducing latency by up to 53%. For single-threaded DBbench, the throughput gains are same as latency reduction gains; hence, we do not show throughput results.

Summary. NoveLSM's immutable memtable reduces write latency by 24% and read latency by up



(a) Write latency (b) Read latency (c) 4 KB Write latency cost (d) 4 KB Read latency cost

Figure 8: **NVM mutability impact.** *Figure shows (a) write latency, (b) read latency. LevelDB-NVM and RocksDB-NVM use NVM for SSTable. LevelDB-NVM [1GB-DRAM] uses a large DRAM memtable; [mutable + para] shows read parallelism. [mutable + para + noflush] shows NoveLSM without persistent flush, [mutable + para + NoSST] shows using only NVM memtable without SSTable. Figure (c) and (d) show NoveLSM Write and Read operation latency cost splitup for 4 KB values.*

to 53%. Lack of direct NVM memtable mutability and frequent compaction impacts write performance.

5.3 NVM Memtable Mutability

To understand the effectiveness of NoveLSM's mutable memtable in reducing compaction cost, we begin with NoveLSM+immut-large discussed in the previous result, and analyze four other NoveLSM techniques: NoveLSM+mutable uses a large (4 GB) NVM memtable which is placed in parallel with the DRAM memtable and allows direct transactional updates (without logging) supported by persistent processor cache flushes; NoveLSM+mutable+para enables read parallelism; NoveLSMmutable+para+noflush shows the latency without persistent processor cache flushes; and finally, NoveLSM+mutable+NoSST uses only persistent NVM memtable for the entire database without SSTables. For comparison, in addition to LevelDB-NVM, we also compare the impact of increasing the vanilla LevelDB-NVM DRAM memtable size to 1GB (LevelDB-NVM+1GB-DRAM) and RocksDB-NVM [3] by placing all its SSTable in NVM. RocksDB-NVM is configured with default configuration values used in other prior work [9, 30]. For our experimentation, we set the DRAM memtable to 64 MB for all configuration except LevelDB-NVM+1GB-DRAM. Figure 8.c and Figure 8.d show the cost split up for a 4 KB random write and read operation.

Write performance. Figure 8.a shows the average write latency as a function of value size. When the mutable NVM memtable is active, its large capacity provides background threads sufficient time to finish compaction, consequently reducing foreground stalls. For 4 KB values, NoveLSM+mutable reduces latency by more than 3.8x compared to LevelDB-NVM and NoveLSM+immut-large, due to reduction of both compaction and log write cost as shown in Figure 8.c. For 64 KB value size, write latency reduces by 2.7x compared to LevelDB-NVM. While increasing the vanilla LevelDB-NVM's DRAM memtable size (1GB-DRAM) improves performance, however, (1) DRAM consumption increases by twice (DRAM memtable and immutable table), (2) increases log size and recovery time (discussed shortly), and importantly, (3) compromises crash consistency because both LevelDB and RocksDB do not commit log updates to storage by default.

For smaller value sizes, RocksDB-NVM marginally reduces write latency compared to mutable NoveLSM (NoveLSM+mutable) design that provides in-place commits (with processor cache flushes). RocksDB benefits come from using a Cuckoo hash-based SST [11] that improves random lookups (but severely impacts scan operations), parallel compaction to exploit SSD parallelism, and not flushing log updates to storage. While incorporating complementary optimizations, such as Cuckoo hash-based SST and parallel compaction, can reduce latency, even avoiding persistent cache flush (NoveLSM+mutable+para+noflush) reduces latency compared to RocksDB. For larger values, NoveLSM reduces latency by 36% compared to RocksDB-NVM providing the same durability guarantees. Finally, NoveLSM+mutable+NoSST, by using large NVM memtable and adding the entire database to the skip list, eliminates compaction cost reducing the write latency by 5x compared to LevelDB-NVM and more than 1.9x compared to RocksDB-NVM.

Read parallelism. Figure 8.b shows the read latency for all configurations. First, compared to LevelDB-NVM, NoveLSM+mutable reduces read latency for 4 KB value size by 30%. RocksDB-NVM with a random-access friendly Cuckoo-hash SSTable significantly reduces memtable miss latency cost, providing better performance for smaller values. For smaller values (1 KB, 4 KB), NoveLSM's optimistic parallelism shows no gains compared to RocksDB because the cost of thread management suppresses benefits of parallel read. However, for larger value sizes, NoveLSM's parallelism combined with the reduction in deserialization cost reduces NoveLSM's read latency by 2x and 24% compared to LevelDB-NVM and RocksDB respectively. In-

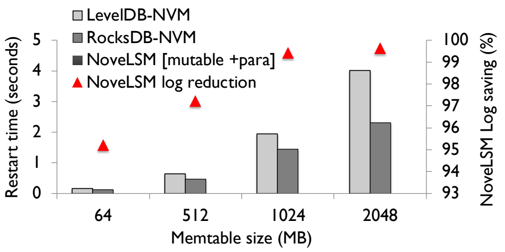


Figure 9: **Failure recovery performance.** *The figure shows recovery time as a function of memtable size in X-axis. For LevelDB and RocksDB, DRAM memtable size is increased, whereas for NoveLSM, NVM memtable increased, and DRAM memtable size is kept constant at 64 MB.*

corporating RocksDB's optimized SSTable can further improve NoveLSM's read performance. As a proof, the NoveLSM-NoSST case reduces the read latency by 45% compared to RocksDB.

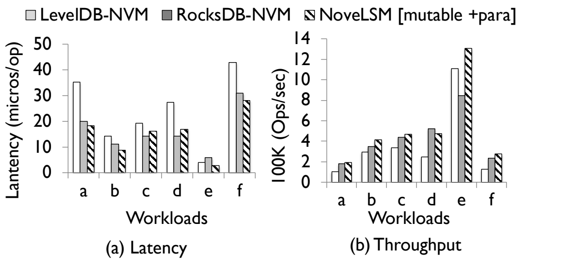
Splitting LSMs across NVM and SSDs. NoveLSM can support large LSMs that spill over to SSD when NVM capacity is full. To understand the performance impact, we set the LSM database size to 16 GB. We compare two approaches: (1) LevelDB-NVM-SSD that splits SSTable across NVM (8 GB) and SSD (8 GB), (2) NoveLSM-mutable-SSD that uses a half-and-half configuration with 4 GB NVM for mutable memtable, 4 GB NVM for higher levels of SSTable, and 8 GB SSTable on SSD. We do not consider RocksDB because of the complexity involved in supporting multiple storage devices for a complex compaction mechanism, which is beyond the scope of this work. When evaluating the two configurations, we determine that LevelDB-NVM-SSD suffers from high compaction cost. For larger value sizes, memtables fill-up quickly, triggering a chain of compaction across both NVM and SSD SSTables. In contrast, NoveLSM's mutable NVM memtable reduces compaction frequency allowing background threads with sufficient time to compact, thus reducing stalls; consequently, NoveLSM reduces latency by more than 45% for 64 KB values compared to LevelDB-NVM-SSD.

Summary. The results highlight the benefits of using a mutable memtable for write operations and supporting parallelism for read operations in both NVM-only and NVM+SSD configurations. Incorporating RocksDB's SSTable optimizations can further improve NoveLSM's performance.

5.4 Failure Recovery

NoveLSM's mutable persistence provides in-place commits to NVM memtable and avoids log updates. In Figure 9, we analyze the impact of memtable size on recovery time after a failure. To emulate failure, we crash DBbench's random write workload after inserting half the keys. On the X-axis, for LevelDB-NVM and RocksDB-NVM, we increase DRAM memtable size, whereas, for NoveLSM-mutable, the DRAM memtable size is kept constant at 64 MB and only the NVM mutable memtable size is varied.

For LevelDB-NVM and RocksDB-NVM, all updates to DRAM memtable are also logged; hence, increasing the DRAM memtable size also increases the log size that must be read during recovery, thereby increasing the recovery time. Recovery involves iterating the log, verifying checksums, serializing logged key-value pairs to an SSTable disk block, and inserting them to the top level of the SSTable which is merged with lower levels. As a result, for a 1 GB DRAM memtable size, LevelDB-NVM's recovery is as high as 4 seconds; RocksDB-NVM recovers faster than LevelDB due to its specialized SST format. For NoveLSM, recovery involves identifying a correct version of the persistent memtable before the crash, memory-mapping the NVM memtable's persistent skip list, and modifying the root pointer to the current virtual address of the application. As a result, restart performance for NoveLSM is more than three orders faster. Importantly, NoveLSM logs only the updates to DRAM memtable, thereby reducing logging writes by up to 99%.



(a) Latency

(b) Throughput

Figure 10: **YCSB (a) latency and (b) throughput.** *Results only shown for run-phase after warm-up. NoveLSM's mutable memtable size set to 4 GB. Workload A has 50-50% update-read ratio, B is read-intensive with 95% reads and 5% updates (overwrites); C is read-only, D is also read-only, with the most recently inserted records being most popular, E is scan-intensive (95% scan, and 5% insert), and F has 50% reads and 50% write-modify-reads.*

5.5 Impact on YCSB

To understand the benefits and implication for cloud workloads, we run the widely-used YCSB [18] benchmark and compare LevelDB-NVM, RocksDB-NVM, and NoveLSM-mutable-para approaches. We use the six workloads from the YCSB cloud suite with different access patterns. YCSB has a warm-up (write-only) and a run phase, and we show the run phase results when using 4-client threads. Figure 10 shows the 95th percentile latency and throughput (in 100K operations per second). We use 4 KB value size and 16 GB database. The SSTables are placed in NVM for all cases, and NoveLSM's mutable memtable is set to 4 GB. First, for workload A, with the highest write ratio

(50%), NoveLSM's direct mutability improves throughput by 6% over RocksDB and 81% over LevelDB-NVM, even for small 4 KB value sizes. Both workload B and workload C are read-intensive, with high random reads. NoveLSM's read parallelism is effective in simultaneously accessing data across multiple LSM levels for four client threads. For workload D, most accesses are recently inserted values, resulting in high mutable and immutable memtable hits even for RocksDB. NoveLSM checks the bloom filter for each access for enabling read parallelism (parallelism is not required as keys are present in memtable), and this check adds around $1\mu s$ overhead per key resulting in a slightly lower throughput compared to RocksDB. Next, for the scan-intensive workload E, LevelDB and NoveLSM's SSTable are highly scan-friendly; in contrast, RocksDB's SSTable optimized for random-access performs poorly for scan operations. As a result, NoveLSM shows 54% higher throughput compared to RocksDB-NVM. Finally, workload F with 50% updates (overwrites) adds significant logging and compaction-related serialization overhead. NoveLSM's direct mutability reduces these cost improving throughput by 17% compared to RocksDB and more than 2x over LevelDB-NVM.

Summary. NoveLSM's direct mutability and read parallelism provide high-performance for both random and sequential workloads.

6 Related Work

Key-value store and storage. Prior works such as SILT [29], FlashStore [20], SkimpyStash [21] design key-value stores specifically targeting SSDs. FlashStore and SkimpyStas treat flash as an intermediate cache and place append-only logs to benefit from the high sequential write performance of SSD. SILT reduces DRAM usage by splitting in-memory log across the DRAM and SSD and maintaining a sorted log index in the memory. In summary, prior works enforce sequentiality by batching and adding software layers for improving throughput. In contrast, we design NoveLSM to reduce I/O access latency with heap-based persistent structures.

Application redesign for persistent memory. Byte addressability, low latency, and high bandwidth make NVMs a popular target for redesigning data structures and applications originally designed for block storage. Venkatraman et al. [36] were one of the first to explore the benefits of persistence-friendly B-trees for NVMs. Since then, several others have redesigned databases [8], key-value stores [31], B-trees [14], and hashtables [19].

LSM and redesign for storage. Several prior works have redesigned LSMs for SSD. Wang et al [39] expose SSD's I/O channel information to LevelDB to exploit the parallel bandwidth usage. WiscKey [30] redesigns LSMs for reducing the read and write amplification and exploiting SSD bandwidth. VT-tree [35] design proposes a file system and a user-level key-value store for workload-independent storage. In NoveLSM, we reduce the write latency with a mutable persistent skip list and the read latency by parallelizing reads across the LSM levels.

LSM redesign for NVM. NoveLSM is focused on extending existing LSMs for NVMs rather than completely redesigning their data structures; this is complementary to projects such as FloDB and PebblesDB [9, 34]. We were recently made aware of a concurrently developed effort with similar goals as NoveLSM. NVM-Rocks [28] shares similar ideas on using a persistent mutable memtable to reduce access latencies and recovery costs. To improve read latencies, it introduces a hierarchy of read caches. NoveLSM retains the in-DRAM memtable of the original LSM design, benefiting latencies for both cached reads and writes, and introduces parallelism *within* read operations to reduce read latency. We look forward to gaining access to NVMrocks and analyzing the tradeoffs that each technique contributes to the overall LSM performance.

7 Conclusion

We present NoveLSM, an LSM-based persistent key-value store that exploits NVM byte-addressability, persistence, and large capacity by designing a heap-based persistent immutable NVM skip list. The immutable NVM skip list facilitates DRAM memtable compaction without incurring memory to I/O data serialization cost and also accelerates reads. To reduce the compaction cost further, we introduce direct mutability of NVM memtables, which allow applications can to directly commit data to NVM memtable with stronger durability and avoid logging. Reducing compaction and logging overheads reduces random write latency by up to 3.8x compared to LevelDB running on NVM. To reduce read latency, we design opportunistic parallelism, which reduces read latency by up to 2x. Finally, the persistent memtable makes the restarts three orders of magnitude faster. As storage moves closer to memory, and storage bottlenecks shifts towards software, increased effort to optimize such software will undoubtedly be required to realize further performance gains.

Acknowledgements

We thank the anonymous reviewers and Michio Honda (our shepherd) for their insightful comments. This material was supported by funding from DoE Office of Science Unity SSIO and NSF grant CNS-1421033. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] Apache Cassandra. http://cassandra.apache.org/.

[2] Apache HBase. http://hbase.apache.org/.

[3] Facebook RocksDB. http://rocksdb.org/.

[4] Google LevelDB . http://tinyurl.com/osqd7c8.

[5] Intel-Micron Memory 3D XPoint. http://intel.ly/1eICR0a.

[6] Linux DAX file system. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[7] ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. *Operating Systems: Three Easy Pieces*, 0.91 ed. Arpaci-Dusseau Books, May 2015.

[8] ARULRAJ, J., PAVLO, A., AND DULLOOR, S. R. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia, 2015), SIGMOD '15.

[9] BALMAU, O., GUERRAOUI, R., TRIGONAKIS, V., AND ZABLOTCHI, I. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia, 2017), EuroSys '17.

[10] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA, 2000), ASPLOS IX.

[11] BORTHAKUR, D. RocksDB Cuckoo SST. http://rocksdb.org/blog/2014/09/12/cuckoo.html.

[12] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2010), MICRO '43.

[13] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008).

[14] CHEN, S., AND JIN, Q. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow. 8*, 7 (Feb. 2015).

[15] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)* (Farmington, PA, November 2013).

[16] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA, 2011), ASPLOS XVI.

[17] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA, 2009), SOSP '09.

[18] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA, 2010), SoCC '10.

[19] DEBNATH, B., HAGHDOOST, A., KADAV, A., KHATIB, M. G., AND UNGUREANU, C. Revisiting Hash Table Design for Phase Change Memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads* (Monterey, California, 2015), INFLOW '15.

[20] DEBNATH, B., SENGUPTA, S., AND LI, J. FlashStore: High Throughput Persistent Key-value Store. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 1414–1425.

[21] DEBNATH, B., SENGUPTA, S., AND LI, J. SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece, 2011), SIGMOD '11.

[22] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands, 2014), EuroSys '14.

[23] GOLAN-GUETA, G., BORTNIKOV, E., HILLEL, E., AND KEIDAR, I. Scaling Concurrent Log-structured Data Stores. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France, 2015), EuroSys '15.

[24] INTEL. Logging library. https://github.com/pmem/nvml.

[25] KANNAN, S., GAVRILOVSKA, A., GUPTA, V., AND SCHWAN, K. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada, 2017), ISCA '17.

[26] KANNAN, S., GAVRILOVSKA, A., AND SCHWAN, K. pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom, 2016), EuroSys '16.

[27] KIM, H., SESHADRI, S., DICKEY, C. L., AND CHIU, L. Phase Change Memory in Enterprise Storage Systems: Silver Bullet or Snake Oil? *SIGOPS Oper. Syst. Rev. 48*, 1 (May 2014), 82–89.

[28] LI, J., PAVLO, A., AND DONG, S. NVMRocks: RocksDB on Non Volatile Memory Systems, 2017. http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/.

[29] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), SOSP '11.

[30] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (Santa Clara, CA, 2016), FAST'16.

[31] MÁRMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. NVMKV: A Scalable and Lightweight Flash Aware Key-value Store. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems* (Philadelphia, PA, 2014), HotStorage'14.

[32] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The Log-structured Merge-tree (LSM-tree). *Acta Inf. 33*, 4 (June 1996).

[33] PUGH, W. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM 33*, 6 (June 1990).

[34] RAJU, P., KADEKODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)* (Shanghai, China, October 2017).

[35] SHETTY, P., SPILLANE, R., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building Workload-independent Storage with VT-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (San Jose, CA, 2013), FAST'13.

[36] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (San Jose, California, 2011), FAST'11.

[37] VOLOS, H., MAGALHAES, G., CHERKASOVA, L., AND LI, J. NVM Quartz emulator. `https://github.com/HewlettPackard/quartz`.

[38] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA, 2011), ASPLOS XVI.

[39] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands, 2014), EuroSys '14.

[40] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based Ultra-large Key-value Store for Small Data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, 2015), USENIX ATC '15.

[41] ZHAO, J., LI, S., YOON, D. H., XIE, Y., AND JOUPPI, N. P. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California, 2013), MICRO-46.

HashKV: Enabling Efficient Updates in KV Storage via Hashing

Helen H. W. Chan[†], Yongkun Li[‡], Patrick P. C. Lee[†], Yinlong Xu[‡]
[†]The Chinese University of Hong Kong [‡]University of Science and Technology of China

Abstract

Persistent key-value (KV) stores mostly build on the Log-Structured Merge (LSM) tree for high write performance, yet the LSM-tree suffers from the inherently high I/O amplification. *KV separation* mitigates I/O amplification by storing only keys in the LSM-tree and values in separate storage. However, the current KV separation design remains inefficient under update-intensive workloads due to its high garbage collection (GC) overhead in value storage. We propose HashKV, which aims for high update performance atop KV separation under update-intensive workloads. HashKV uses *hash-based data grouping*, which deterministically maps values to storage space so as to make both updates and GC efficient. We further relax the restriction of such deterministic mappings via simple but useful design extensions. We compare HashKV with state-of-the-art KV stores via extensive testbed experiments, and show that HashKV achieves 4.6× throughput and 53.4% less write traffic compared to the current KV separation design.

1 Introduction

Persistent key-value (KV) stores are an integral part of modern large-scale storage infrastructures for storing massive structured data (e.g., [4, 6, 10, 18]). While real-world KV storage workloads are mainly read-intensive (e.g., the Get/Update ratio can reach 30:1 in Facebook's Memcached workloads [3]), *update-intensive* workloads are also dominant in many storage scenarios, including online transaction processing [37] and enterprise servers [17]. For example, Yahoo! reports that its low-latency workloads increasingly move from reads to writes [33].

Modern KV stores optimize the performance of writes (including inserts and updates) using the Log-Structured Merge (LSM) tree [29]. Its idea is to maintain sequentiality of random writes through a log-structured (append-only) design [31], while supporting efficient queries including individual key lookups and range scans. In a nutshell, the LSM-tree buffers written KV pairs and flushes them into a multi-level tree, in which each node is a fixed-size file containing sorted KV pairs and their metadata. The recently written KV pairs are stored at higher tree levels, and are merged with lower tree levels via *compaction*. The LSM-tree design not only improves write performance by avoiding small random updates (which are also harmful to the endurance of solid-state drives (SSDs) [2, 27]), but also improves range scan performance by holding sorted KV pairs in each node.

However, the LSM-tree incurs high I/O amplification in both writes and reads. As the LSM-tree receives more writes of KV pairs, it will trigger frequent compaction operations, leading to tremendous extra I/Os due to rewrites across levels. Such *write amplification* can reach a factor of at least 50× [23, 39], which is detrimental to both write performance and the endurance of SSDs [2, 27]. Also, as the LSM-tree grows in size, reading the KV pairs at lower levels incurs many disk accesses. Such read amplification can reach a factor of over 300× [23], leading to low read performance.

In order to mitigate the compaction overhead, many research efforts focus on optimizing LSM-tree indexing (see §5). One approach is *KV separation* from WiscKey [23], in which keys and metadata are still stored in the LSM-tree, while values are separately stored in an append-only circular log. The main idea of KV separation is to reduce the LSM-tree size, while preserving the indexing feature of the LSM-tree for efficient inserts/updates, individual key lookups, and range scans.

In this work, we argue that KV separation itself still cannot fully achieve high performance under update-intensive workloads. The root cause is that the circular log for value storage needs frequent garbage collection (GC) to reclaim the space from the KV pairs that are deleted or superseded by new updates. However, the GC overhead is actually expensive due to two constraints of the circular log. First, the circular log maintains a strict GC order, as it always performs GC at the beginning of the log where the least recently written KV pairs are located. This can incur a large amount of unnecessary data relocation (e.g., when the least recently written KV pairs remain valid). Second, the GC operation needs to query the LSM-tree to check the validity of each KV pair. These queries have high latencies, especially when the LSM-tree becomes sizable under large workloads.

We propose HashKV, a high-performance KV store tailored for update-intensive workloads. HashKV builds on KV separation and uses a novel *hash-based data grouping* design for value storage. Its idea is to divide value storage into fixed-size partitions and deterministically map the value of each written KV pair to a partition by hashing its key. Hash-based data grouping supports lightweight updates due to deterministic mapping. More importantly, it significantly mitigates GC overhead, since each GC operation not only has the flexibility to select a partition to reclaim space, but also eliminates the queries to the LSM-tree for checking the validity of KV pairs.

On the other hand, the deterministic nature of hash-based data grouping restricts where KV pairs are stored. Thus, we propose three novel design extensions to relax the restriction of hash-based data grouping: (i) *dynamic reserved space allocation*, which dynamically allocates reserved space for extra writes if their original hash partitions are full given the size limit; (ii) *hotness awareness*, which separates the storage of hot and cold KV pairs to improve GC efficiency as inspired by existing SSD designs [19,27]; and (iii) *selective KV separation*, which keeps small-size KV pairs in entirety in the LSM-tree to simplify lookups.

We implement our HashKV prototype atop LevelDB [15], and show via testbed experiments that HashKV achieves 4.6× throughput and 53.4% less write traffic compared to the circular log design in WiscKey under update-intensive workloads. Also, HashKV generally achieves higher throughput and significantly less write traffic compared to modern KV stores, such as LevelDB and RocksDB [12], in various cases.

Our work makes a case of augmenting KV separation with a new value management design. While the key and metadata management of HashKV now builds on LevelDB, it can also adopt other KV stores with new LSM tree designs (e.g., [30,33,35,39,41,42]). How HashKV affects the performance of various LSM-tree-based KV stores under KV separation is posed as future work. The source code of HashKV is available for download at: **http://adslab.cse.cuhk.edu.hk/software/hashkv**.

2 Motivation

We use LevelDB [15] as a representative example to explain the write and read amplification problems of LSM-tree-based KV stores. We show how KV separation [23] mitigates both write and read amplifications, yet it still cannot fully achieve efficient updates.

2.1 LevelDB

LevelDB organizes KV pairs based on the LSM-tree [29], which transforms small random writes into sequential writes and hence maintains high write performance. Figure 1 illustrates the data organization in LevelDB. It divides the storage space into k levels (where $k > 1$) denoted by L_0, L_1, $\cdots$, L_{k-1}. It configures the capacity of each level L_i to be a multiple (e.g., 10×) of that of its upper level L_{i-1} (where $1 \leq i \leq k-1$).

For inserts or updates of KV pairs, LevelDB first stores the new KV pairs in a fixed-size in-memory buffer called *MemTable*, which uses a skip-list to keep all buffered KV pairs sorted by keys. When the MemTable is full, LevelDB makes it *immutable* and flushes it to disk at level L_0 as a file called *SSTable*. Each SSTable has a size of around 2 MiB and is also immutable. It stores indexing metadata, a Bloom filter (for quickly checking if a KV pair exists in the SSTable), and all sorted KV pairs.

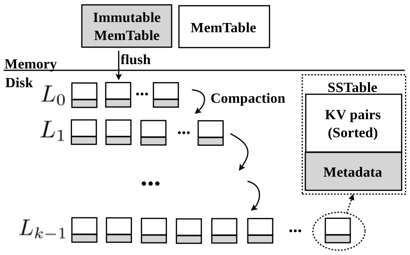


Figure 1: Data organization in LevelDB.

If L_0 is full, LevelDB flushes and merges its KV pairs into L_1 via *compaction*; similarly, if L_1 is full, its KV pairs are flushed and merged into L_2, and so on. The compaction process comprises three steps. First, it reads out KV pairs in both L_i and L_{i+1} into memory (where $i \geq 0$). Second, it sorts the valid KV pairs (i.e., newly inserted or updated) by keys and reorganizes them into SSTables. It also discards all invalid KV pairs (i.e., deleted or superseded by new updates). Finally, it writes back all SSTables with valid KV pairs to L_{i+1}. Note that all KV pairs in each level, except L_0, are sorted by keys. In L_0, LevelDB only keeps KV pairs sorted within each SSTable, but not across SSTables. This improves performance of flushing from the MemTable to disk.

To perform a key lookup, LevelDB searches from L_0 to L_{k-1} and returns the first associated value found. In L_0, LevelDB searches all SSTables. In each level between L_1 and L_{k-1}, LevelDB first identifies a candidate SSTable and checks the Bloom filter in the candidate SSTable to determine if the KV pair exists. If so, LevelDB reads the SSTable file and searches for the KV pair; otherwise, it directly searches the lower levels.

Limitations: LevelDB achieves high random write performance via the LSM-tree-based design, but suffers from both *write and read amplifications*. First, the compaction process inevitably incurs extra reads and writes. In the worst case, to merge one SSTable from L_{i-1} to L_i, it reads and sorts 10 SSTables, and writes back all SSTables. Prior studies show that LevelDB can have an overall write amplification of at least 50× [23,39], since it may trigger more than one compaction to move a KV pair down multiple levels under large workloads.

In addition, a lookup operation may search multiple levels for a KV pair and incur multiple disk accesses. The reason is that the search in each level needs to read the indexing metadata and the Bloom filter in the associated SSTable. Although the Bloom filter is used, it may introduce false positives. In this case, an SSTable is still unnecessarily read from disk even though the KV pair actually does not exist. Thus, each lookup typically incurs multiple disk accesses. Such read amplification further aggravates under large workloads, as the LSM-tree builds up in levels. Measurements show that the read amplification reaches over 300× in the worst case [23].

2.2 KV Separation

KV separation, proposed by WiscKey [23], decouples the management of keys and values to mitigate both write and read amplifications. The rationale is that storing values in the LSM-tree is unnecessary for indexing. Thus, WiscKey stores only keys and metadata (e.g., key/value sizes, value locations, etc.) in the LSM-tree, while storing values in a separate append-only, circular log called *vLog*. KV separation effectively mitigates write and read amplifications of LevelDB as it significantly reduces the size of the LSM-tree, and hence both compaction and lookup overheads.

Since vLog follows the log-structured design [31], it is critical for KV separation to achieve lightweight *garbage collection (GC)* in vLog, i.e., to reclaim the free space from invalid values with limited overhead. Specifically, WiscKey tracks the *vLog head* and the *vLog tail*, which correspond to the end and the beginning of vLog, respectively. It always inserts new values to the vLog head. When it performs a GC operation, it reads a chunk of KV pairs from the vLog tail. It first queries the LSM-tree to see if each KV pair is valid. It then discards the values of invalid KV pairs, and writes back the valid values to the vLog head. It finally updates the LSM-tree for the latest locations of the valid values. To support efficient LSM-tree queries during GC, WiscKey also stores the associated key and metadata together with the value in vLog. Note that vLog is often over-provisioned with extra reserved space to mitigate GC overhead.

Limitations: While KV separation reduces compaction and lookup overheads, we argue that it suffers from the substantial GC overhead in vLog. Also, the GC overhead becomes more severe if the reserved space is limited. The reasons are two-fold.

First, vLog can only reclaim space from its vLog tail due to its circular log design. This constraint may incur unnecessary data movements. In particular, real-world KV storage often exhibits strong locality [3], in which a small portion of *hot* KV pairs are frequently updated, while the remaining *cold* KV pairs receive only few or even no updates. Maintaining a strict sequential order in vLog inevitably relocates cold KV pairs many times and increases GC overhead.

Also, each GC operation queries the LSM-tree to check the validity of each KV pair in the chunk at the vLog tail. Since the keys of the KV pairs may be scattered across the entire LSM-tree, the query overhead is high and increases the latency of the GC operation. Even though KV separation has already reduced the size of the LSM-tree, the LSM-tree is still sizable under large workloads, and this aggravates the query cost.

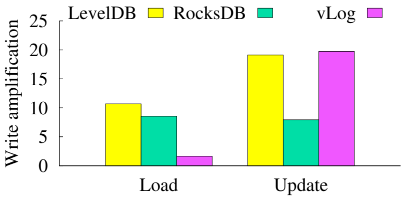


Figure 2: Write amplifications of LevelDB, RocksDB, and vLog in the Load and Update phases.

To validate the limitations of KV separation, we implement a KV store prototype based on vLog (see §3.8) and evaluate its write amplification. We consider two phases: Load and Update. In the Load phase, we insert 40 GiB of 1-KiB KV pairs into vLog that is initially empty; in the Update phase, we issue 40 GiB of updates to the existing KV pairs based on a Zipf distribution with a Zipfian constant of 0.99. We provision 40 GiB of space for vLog, and an additional 30% (12 GiB) of reserved space. We also disable the write cache in our prototype (see §3.2). Figure 2 shows the write amplification results of vLog in the Load and Update phases, in terms of the ratio of the total device write size to the actual write size due to inserts or updates. For comparison, we also consider two modern KV stores, LevelDB [15] and RocksDB [12], based on their default parameters. In the Load phase, vLog has sufficient space to hold all KV pairs and does not trigger GC, so its write amplification is only 1.6× due to KV separation. However, in the Update phase, the updates fill up the reserved space and start to trigger GC. We see that vLog has a write amplification of 19.7×, which is close to LevelDB (19.1×) and higher than RocksDB (7.9×).

To mitigate GC overhead in vLog, one approach is to partition vLog into segments and choose the best candidate segments that minimize GC overhead based on the cost-benefit policy or its variants [26, 31, 32]. However, the hot and cold KV pairs can still be mixed together in vLog, so the chosen segments for GC may still contain cold KV pairs that are unnecessarily moved.

To address the mixture of hot and cold data, a better approach is to perform *hot-cold data grouping* as in SSD designs [19, 27], in which we separate the storage of hot and cold KV pairs into two regions and apply GC to each region individually (more GC operations are expected to be applied to the storage region for hot KV pairs). However, the direct implementation of hot-cold data grouping inevitably increases the update latency in KV separation. As a KV pair may be stored in either hot or cold regions, each update needs to first query the LSM-tree for the exact storage location of the KV pair. Thus, a key motivation of our work is to enable hotness awareness without LSM-tree lookups.

3 HashKV Design

HashKV is a persistent KV store that specifically targets update-intensive workloads. It improves the management of value storage atop KV separation to achieve high update performance. It supports standard KV operations: PUT (i.e., writing a KV pair), GET (i.e., retrieving the value of a key), DELETE (i.e., deleting a KV pair), and SCAN (i.e., retrieving the values of a range of keys).

3.1 Main Idea

HashKV follows KV separation [23] by storing only keys and metadata in the LSM-tree for indexing KV pairs, while storing values in a separate area called the *value store*. Atop KV separation, HashKV introduces several core design elements to achieve efficient value storage management.

- **Hash-based data grouping:** Recall that vLog incurs substantial GC overhead in value storage. Instead, HashKV maps values into fixed-size partitions in the value store by hashing the associated keys. This design achieves: (i) *partition isolation*, in which all versions of value updates associated with the same key must be written to the same partition, and (ii) *deterministic grouping*, in which the partition where a value should be stored is determined by hashing. We leverage this design to achieve flexible and lightweight GC.
- **Dynamic reserved space allocation:** Since we map values into fixed-size partitions, one challenge is that a partition may receive more updates than it can hold. HashKV allows a partition to grow *dynamically* beyond its size limit by allocating fractions of reserved space in the value store.
- **Hotness awareness:** Due to deterministic grouping, a partition may be filled with the values from a mix of hot and cold KV pairs, in which case a GC operation unnecessarily reads and writes back the values of cold KV pairs. HashKV uses a *tagging* approach to relocate the values of cold KV pairs to a different storage area and separate the hot and cold KV pairs, so that we can apply GC to hot KV pairs only and avoid re-copying cold KV pairs.
- **Selective KV separation:** HashKV differentiates KV pairs by their value sizes, such that the small-size KV pairs can be directly stored in the LSM-tree without KV separation. This saves the overhead of accessing both the LSM-tree and the value store for small-size KV pairs, while the compaction overhead of storing the small-size KV pairs in the LSM-tree is limited.

Remarks: HashKV maintains a single LSM-tree for indexing (instead of hash-partitioning the LSM-tree as in the value store) to preserve the ordering of keys and the range scan performance. Since hash-based data grouping spreads KV pairs across the value store, it incurs random writes; in contrast, vLog maintains sequential writes with a log-structured storage layout. Our HashKV prototype (see §3.8) exploits both multi-threading and batch writes to limit random write overhead.

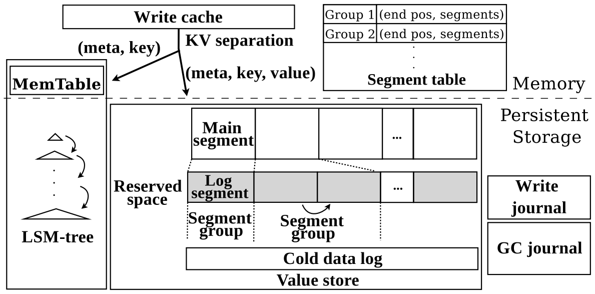


Figure 3: HashKV architecture.

3.2 Storage Management

Figure 3 depicts the architecture of HashKV. It divides the logical address space of the value store into fixed-size units called *main segments*. Also, it over-provisions a fixed portion of reserved space, which is again divided into fixed-size units called *log segments*. Note that the sizes of main segments and log segments may differ; by default, we set them as 64 MiB and 1 MiB, respectively.

For each insert or update of a KV pair, HashKV hashes its key into one of the main segments. If the main segment is not full, HashKV stores the value in a log-structured manner by appending the value to the end of the main segment; on the other hand, if the main segment is full, HashKV dynamically allocates a free log segment to store the extra values in a log-structured manner. Again, it further allocates additional free log segments if the current log segment is full. We collectively call a main segment and all its associated log segments a *segment group*. Also, HashKV updates the LSM-tree for the latest value location. To keep track of the storage status of the segment groups and segments, HashKV uses a global in-memory *segment table* to store the current end position of each segment group for subsequent inserts or updates, as well as the list of log segments associated with each segment group. Our design ensures that each insert or update can be directly mapped to the correct write position without issuing LSM-tree lookups on the write path, thereby achieving high write performance. Also, the updates of the values associated with the same key must go to the same segment group, and this simplifies GC. For fault tolerance, HashKV checkpoints the segment table to persistent storage.

To facilitate GC, HashKV also stores the key and metadata (e.g., key/value sizes) together with the value for each KV pair in the value store as in WiscKey [23] (see Figure 3). This enables a GC operation to quickly identify the key associated with a value when it scans the value store. However, our GC design inherently differs from vLog used by WiscKey (see §3.3).

To improve write performance, HashKV holds an in-memory *write cache* to store the recently written KV pairs, at the expense of degrading reliability. If the key of a new KV pair to be written is found in the write cache, HashKV directly updates the value of the cached key in-place without issuing the writes to the LSM-tree and the value store. It can also return the KV pairs from the write cache for reads. If the write cache is full, HashKV flushes all the cached KV pairs to the LSM-tree and the value store. Note that the write cache is an optional component and can be disabled for reliability concerns.

HashKV supports hotness awareness by keeping cold values in a separate *cold data log* (see §3.4). It also addresses crash consistency by tracking the updates in both *write journal* and *GC journal* (see §3.7).

3.3 Garbage Collection (GC)

HashKV necessitates GC to reclaim the space occupied by invalid values in the value store. In HashKV, GC operates in units of segment groups, and is triggered when the free log segments in the reserved space are running out. At a high level, a GC operation first selects a candidate segment group and identifies all valid KV pairs (i.e., the KV pairs of the latest version) in the group. It then writes back all valid KV pairs to the main segment, or additional log segments if needed, in a log-structured manner. It also releases any unused log segments that can be later used by other segment groups. Finally, it updates the latest value locations in the LSM-tree. Here, the GC operation needs to address two issues: (i) which segment group should be selected for GC; and (ii) how the GC operation quickly identifies the valid KV pairs in the selected segment group.

Unlike vLog, which requires the GC operation to follow a strict sequential order, HashKV can flexibly choose which segment group to perform GC. It currently adopts a *greedy* approach and selects the segment group with the largest amount of writes. Our rationale is that the selected segment group typically holds the hot KV pairs that have many updates and hence has a large amount of writes. Thus, selecting this segment group for GC likely reclaims the most free space. To realize the greedy approach, HashKV tracks the amount of writes for each segment group in the in-memory segment table (see §3.2), and uses a *heap* to quickly identify which segment group receives the largest amount of writes.

To check the validity of KV pairs in the selected segment group, HashKV sequentially scans the KV pairs in the segment group without querying the LSM-tree (note that it also checks the write cache for any latest KV pairs in the segment group). Since the KV pairs are written to the segment group in a log-structured manner, the KV pairs must be sequentially placed according to their order of being updated. For a KV pair that has multiple versions of updates, the version that is nearest to the end of the segment group must be the latest one and correspond to the valid KV pair, while other versions are invalid. Thus, the running time for each GC operation only depends on the size of the segment group that needs to be scanned. In contrast, the GC operation in vLog reads a chunk of KV pairs from the vLog tail (see §2.2). It queries the LSM-tree (based on the keys stored along with the values) for the latest storage location of each KV pair in order to check if the KV pair is valid [23]. The overhead of querying the LSM-tree becomes substantial under large workloads.

During a GC operation on a segment group, HashKV constructs a temporary in-memory hash table (indexed by keys) to buffer the addresses of the valid KV pairs being found in the segment group. As the key and address sizes are generally small and the number of KV pairs in a segment group is limited, the hash table has limited size and can be entirely stored in memory.

3.4 Hotness Awareness

Hot-cold data separation improves GC performance in log-structured storage (e.g., SSDs [19, 27]). In fact, the current hash-based data grouping design realizes some form of hot-cold data separation, since the updates of the hot KV pairs must be hashed to the same segment group and our current GC policy always chooses the segment group that is likely to store the hot KV pairs (see §3.3). However, it is inevitable that some cold KV pairs are hashed to the segment group selected for GC, leading to unnecessary data rewrites. Thus, a challenge is to fully realize hot-cold data separation to further improve GC performance.

HashKV relaxes the restriction of hash-based data grouping via a *tagging* approach (see Figure 4). Specifically, when HashKV performs a GC operation on a segment group, it classifies each KV pair in the segment group as hot or cold. Currently, we treat the KV pairs that are updated at least once since their last inserts as hot, or cold otherwise (more accurate hot-cold data identification approaches [16] can be used). For the hot KV pairs, HashKV still writes back their latest versions to the same segment group via hashing. However, for the cold KV pairs, it now writes their values to a separate storage area, and keeps their metadata only (i.e., without values) in the segment group. In addition, it adds a *tag* in the metadata of each cold KV pair to indicate its presence in the segment group. Thus, if a cold KV pair is later updated, we know directly from the tag (without querying the LSM-tree) that the cold KV pair has already been stored, so that we can treat it as hot based on our classification policy; the tagged KV pair will also become invalid. Finally, at the end of the GC operation, HashKV updates the latest value locations in the LSM-

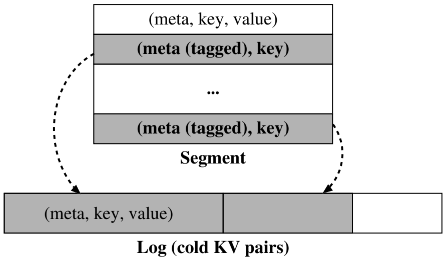


Figure 4: Tagging in HashKV.

tree, such that the locations of the cold KV pairs point to the separate area.

With tagging, HashKV avoids storing the values of cold KV pairs in the segment group and rewriting them during GC. Also, tagging is only triggered during GC, and does not add extra overhead to the write path. Currently, we implement the separate storage area for cold KV pairs as an append-only log (called the *cold data log*) in the value store, and perform GC on the cold data log as in vLog. The cold data log can also be put in secondary storage with a larger capacity (e.g., hard disks) if the cold KV pairs are rarely accessed.

3.5 Selective KV Separation

HashKV supports workloads with general value sizes. Our rationale is that KV separation reduces compaction overhead especially for large-size KV pairs, yet its benefits for small-size KV pairs are limited, and it incurs extra overhead of accessing both the LSM-tree and the value store. Thus, we propose *selective* KV separation, in which we still apply KV separation to KV pairs with large value sizes, while storing KV pairs with small value sizes in *entirety* in the LSM-tree. A key challenge of selective KV separation is to choose the KV pair size threshold of differentiating between small-size and large-size KV pairs (assuming that the key size remains fixed). We argue that the choice depends on the deployment environment. In practice, we can conduct performance tests for different value sizes to see when the throughput gain of selective KV separation becomes significant.

3.6 Range Scans

One critical reason of using the LSM-tree for indexing is its efficient support of range scans. Since the LSM-tree stores and sorts KV pairs by keys, it can return the values of a range of keys via sequential reads. However, KV separation now stores values in separate storage space, so it incurs extra reads of values. In HashKV, the values are scattered across different segment groups, so range scans will trigger many random reads that degrade performance. HashKV currently leverages the *read-ahead* mechanism to speed up range scans by prefetching values into the page cache. For each scan request, HashKV iterates over the range of sorted keys in the LSM-tree, and issues a read-ahead request to each value (via `posix_fadvise`). It then reads all values and returns the sorted KV pairs.

3.7 Crash Consistency

Crashes can occur while HashKV issues writes to persistent storage. HashKV addresses crash consistency based on *metadata journaling* and focuses on two aspects: (i) flushing the write cache and (ii) GC operations.

Flushing the write cache involves writing the KV pairs to the value store and updating metadata in the LSM-tree. HashKV maintains a *write journal* to track each flushing operation. It performs the following steps when flushing the write cache: (i) flushing the cached KV pairs to the value store; (ii) appending metadata updates to the write journal; (iii) writing a commit record to the journal end; (iv) updating keys and metadata in the LSM-tree; and (v) marking the flush operation free in the journal (the freed journaling records can be recycled later). If a crash occurs after step (iii) completes, HashKV replays the updates in the write journal and ensures that the LSM-tree and the value store are consistent.

Handling crash consistency in GC operations is different, as they may overwrite existing valid KV pairs. Thus, we also need to protect existing valid KV pairs against crashes during GC. HashKV maintains a *GC journal* to track each GC operation. It performs the following steps after identifying all valid KV pairs during a GC operation: (i) appending the valid KV pairs that are overwritten as well as metadata updates to the GC journal; (ii) writing all valid KV pairs back to the segment group; (iii) updating the metadata in the LSM-tree; and (iv) marking the GC operation free in the journal.

3.8 Implementation Details

We prototype HashKV in C++ on Linux. We use LevelDB v1.20 [15] for the LSM-tree. Our prototype contains around 6.7K lines of code (without LevelDB).

Storage organization: We currently deploy HashKV on a RAID array with multiple SSDs for high I/O performance. We create a software RAID volume using `mdadm` [22], and mount the RAID volume as an Ext4 file system, on which we run both LevelDB and the value store. In particular, HashKV manages the value store as a large file. It partitions the value store file into two regions, one for main segments and another for log segments, according to the pre-configured segment sizes. All segments are aligned in the value store file, such that the start offset of each main (resp. log) segment is a multiple of the main (resp. log) segment size. If hotness awareness is enabled (see §3.4), HashKV adds a separate region in the value store file for the cold data log. Also, to address crash consistency (see §3.7), HashKV uses separate files to store both write and GC journals.

Multi-threading: HashKV implements multi-threading via `threadpool` [36] to boost I/O performance when flushing KV pairs in the write cache to different segments (see §3.2) and retrieving segments from segment groups in parallel during GC (see §3.3).

To mitigate random write overhead due to deterministic grouping (see §3.1), HashKV implements batch writes. When HashKV flushes KV pairs in the write cache, it first identifies and buffers a number of KV pairs that are hashed to the same segment group in a *batch*, and then issues a sequential write (via a thread) to flush the batch. A larger batch size reduces random write overhead, yet it also degrades parallelism. Currently, we configure a batch write threshold, such that after adding a KV pair into a batch, if the batch size reaches or exceeds the batch size threshold, the batch will be flushed; in other words, HashKV directly flushes a KV pair if its size is larger than the batch write threshold.

4 Evaluation

We compare via testbed experiments HashKV with several state-of-the-art KV stores: LevelDB (v1.20) [15], RocksDB (v5.8) [12], HyperLevelDB [11], PebblesDB [30], and our own vLog implementation for KV separation based on WiscKey [23]. For fair comparison, we build a unified framework to integrate such systems and HashKV. Specifically, all written KV pairs are buffered in the write cache and flushed when the write cache is full. For LevelDB, RocksDB, HyperLevelDB, and PebblesDB, we flush all KV pairs in entirety to them; for vLog and HashKV, we flush keys and metadata to LevelDB, and values (together with keys and metadata) to the value store. We address the following questions:

- How is the update performance of HashKV compared to other KV stores under update-intensive workloads? (Experiment 1)
- How do the reserved space size and RAID configurations affect the update performance of HashKV? (Experiments 2 and 3)
- What is the performance of HashKV under different workloads (e.g., varying KV pair sizes and range scans)? (Experiments 4 and 5)
- What are the performance gains of hotness awareness and selective KV separation? (Experiments 6 and 7)
- How does the crash consistency mechanism affect the update performance of HashKV? (Experiment 8)
- How do parameter configurations (e.g., main segment size, log segment size, and write cache size) affect the update performance of HashKV? (Experiment 9)

In our technical report [5], we present results of additional experiments on the storage space usage, update performance, and range scan performance of HashKV and state-of-the-art KV stores.

4.1 Setup

Testbed: We conduct our experiments on a machine running Ubuntu 14.04 LTS with Linux kernel 3.13.0. The machine is equipped with a quad-core Xeon E3-1240v2, 16 GiB RAM, and seven Plextor M5 Pro 128 GiB SSDs. We attach one SSD to the motherboard as the OS drive, and attach six SSDs to the LSI SAS 9201-16i host bus adapter to form a RAID volume (with a chunk size of 4 KiB) for the KV stores (see §3.8).

Default setup: For LevelDB, RocksDB, HyperLevelDB, and PebblesDB, we use their default parameters. We allow them to use all available capacity in our SSD RAID volume, so that their major overheads come from read and write amplifications in the LSM-tree management.

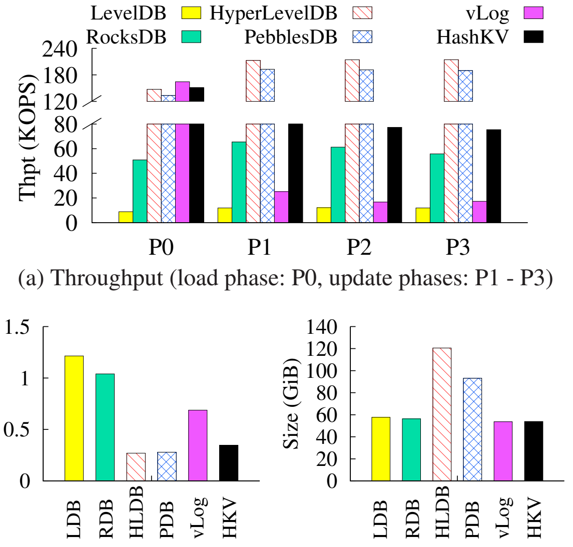
For vLog, we configure it to read 64 MiB from the vLog tail (see §2.2) in each GC operation. For HashKV, we set the main segment size as 64 MiB and the log segment size as 1 MiB. Both vLog and HashKV are configured with 40 GiB of storage space and over-provisioned with 30% (or 12 GiB) of reserved space, while their key and metadata storage in LevelDB can use all available storage space. Here, we provision the storage space of vLog and HashKV to be close to the actual KV store sizes of LevelDB and RocksDB based on our evaluation (see Experiment 1).

We mount the SSD RAID volume under RAID-0 (no fault tolerance) by default to maximize performance. All KV stores run in asynchronous mode and are equipped with a write cache of size 64 MiB. For HashKV, we set the batch write threshold (see §3.8) to 4 KiB, and configure 32 and 8 threads for write cache flushing and segment retrieval in GC, respectively. We disable selective KV separation, hotness awareness, and crash consistency in HashKV by default, except when we evaluate them.

4.2 Performance Comparison

We compare the performance of different KV stores under update-intensive workloads. Specifically, we generate workloads using YCSB [7], and fix the size of each KV pair as 1 KiB, which consists of the 8-B metadata (including the key/value size fields and reserved information), 24-B key, and 992-B value. We assume that each KV store is initially empty. We first load 40 GiB of KV pairs (or 42 M inserts) into each KV store (call it Phase P0). We then repeatedly issue 40 GiB of updates over the existing 40 GiB of KV pairs *three* times (call them Phases P1, P2, and P3), accounting for 120 GiB or 126 M updates in total. Updates in each phase follow a heavy-tailed Zipf distribution with a Zipfian constant of 0.99. We issue the requests to each KV store as fast as possible to stress-test its performance.

Note that vLog and HashKV do not trigger GC in P0. In P1, when the reserved space becomes full after 12 GiB of updates, both systems start to trigger GC; in both P2

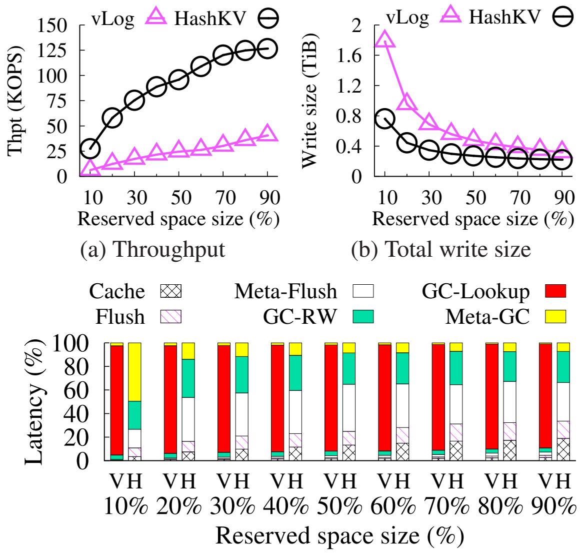


(a) Throughput (load phase: P0, update phases: P1 - P3)

(b) Total write size

(c) KV store size

Figure 5: Experiment 1: Performance comparison of KV stores under update-intensive workloads.



(a) Throughput

(b) Total write size

(c) Latency breakdown (‘V’= vLog; ‘H’= HashKV)

Figure 6: Experiment 2: Impact of reserved space size.

and P3, updates are issued to the fully filled value store and will trigger GC frequently. We include both P2 and P3 to ensure that the update performance is stable.

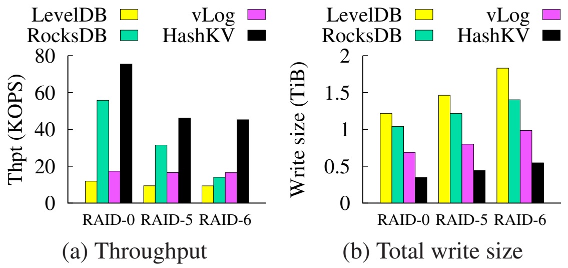
Experiment 1 (Load and update performance): We evaluate LevelDB (LDB), RocksDB (RDB), HyperLevelDB (HDB), PebblesDB (PDB), vLog, and HashKV (HKV), under update-intensive workloads. We first compare LevelDB, RocksDB, vLog, and HashKV; later, we also include HyperLevelDB and PebblesDB into our comparison.

Figure 5(a) shows the performance of each phase. For vLog and HashKV, the throughput in the load phase is higher than those in the update phases, as the latter is dominated by the GC overhead. In the load phase, the throughput of HashKV is 17.1× and 3.0× over LevelDB and RocksDB, respectively. HashKV's throughput is 7.9% slower than vLog, due to random writes introduced to distribute KV pairs via hashing. In the update phases, the throughput of HashKV is 6.3-7.9×, 1.3-1.4×, and 3.7-4.6× over LevelDB, RocksDB, and vLog, respectively. LevelDB has the lowest throughput among all KV stores due to significant compaction overhead, while vLog also suffers from high GC overhead.

Figures 5(b) and 5(c) show the total write sizes and the KV store sizes of different KV stores after all load and update requests are issued. HashKV reduces the total write sizes of LevelDB, RocksDB and vLog by 71.5%, 66.7%, and 49.6%, respectively. Also, they have very similar KV store sizes.

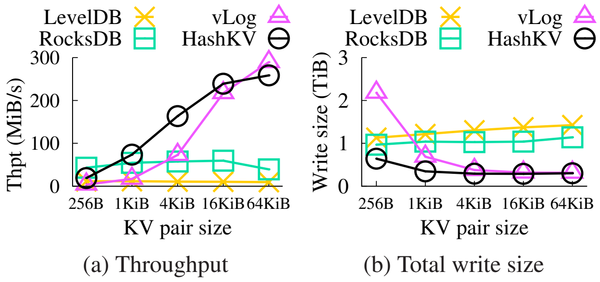
For HyperLevelDB and PebblesDB, both of them have high load and update throughput due to their low compaction overhead. For example, PebblesDB appends fragmented SSTables from the higher level to the lower level, without rewriting SSTables at the lower level [30]. Both HyperLevelDB and PebblesDB achieve at least twice throughput of HashKV, while incurring lower write sizes than HashKV. On the other hand, they incur significant storage overhead, and their final KV store sizes are 2.2× and 1.7× over HashKV, respectively. The main reason is that both HyperLevelDB and PebblesDB compact only selected ranges of keys to reduce write amplification, such that there may still remain many invalid KV pairs after compaction. They also trigger compaction operations less frequently than LevelDB. Both factors lead to high storage overhead. We provide more detailed analysis on the high storage costs of HyperLevelDB and PebblesDB in [5]. In the following experiments, we focus on LevelDB, RocksDB, vLog, and HashKV, as they have comparable storage overhead.

Experiment 2 (Impact of reserved space): We study the impact of reserved space size on the update performance of vLog and HashKV. We vary the reserved space size from 10% to 90% (of 40 GiB). Figure 6 shows the performance in Phase P3, including the update throughput, the total write size, and the latency breakdown. Both vLog and HashKV benefit from the increase in reserved space. Nevertheless, HashKV achieves 3.1-4.7× throughput of vLog and reduces the write size of vLog by 30.1-57.3% across different reserved space sizes. As shown in Figure 6(c), the queries to the LSM-tree during GC incur substantial performance overhead to vLog. We observe that HashKV spends less time on updating metadata during GC (“Meta-GC”) in the LSM-tree with the increasing reserved space size due to less frequent GC operations.



(a) Throughput (b) Total write size

Figure 7: Experiment 3: Different RAID configurations.



(a) Throughput (b) Total write size

Figure 8: Experiment 4: Performance of KV stores under different KV pair sizes.

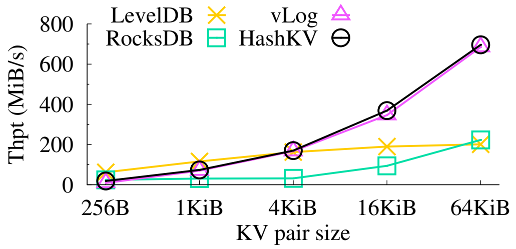


Figure 9: Experiment 5: Range scan performance of different KV stores.

Experiment 3 (Impact of parity-based RAID): We evaluate the impact of the fault tolerance configuration of RAID on the update performance of LevelDB, RocksDB, vLog, and HashKV. We configure the RAID volume to run two parity-based RAID schemes, RAID-5 (single-device fault tolerance) and RAID-6 (double-device fault tolerance). We include the results under RAID-0 for comparison. Figure 7 shows the throughput in Phase P3 and the total write size. RocksDB and HashKV are more sensitive to RAID configurations (larger drops in throughput), since their performance is write-dominated. Nevertheless, the throughput of HashKV is higher than other KV stores under parity-based RAID schemes, e.g., 4.8×, 3.2×, and 2.7× over LevelDB, RocksDB, and vLog, respectively, under RAID-6. The write sizes of KV stores under RAID-5 and RAID-6 increase by around 20% and 50%, respectively, compared to RAID-0, which match the amount of redundancy of the corresponding parity-based RAID schemes.

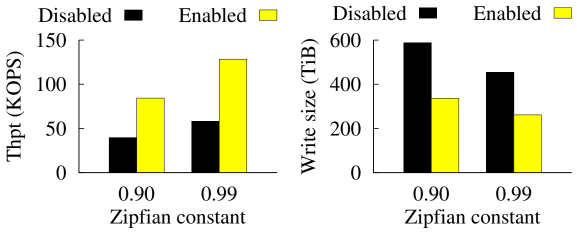
4.3 Performance under Different Workloads

We now study the update and range scan performance of HashKV for different KV pair sizes.

Experiment 4 (Impact of KV pair size): We study the impact of KV pair sizes on the update performance of KV stores. We vary the KV pair size from 256 B to 64 KiB. Specifically, we increase the KV pair size by increasing the value size and keeping the key size fixed at 24 B. We also reduce the number of KV pairs loaded or updated, such that the total size of KV pairs is fixed at 40 GiB. Figure 8 shows the update performance of KV stores in Phase P3 versus the KV pair size. The throughput of LevelDB and RocksDB remains similar across most KV pair sizes, while the throughput of vLog and HashKV increases as the KV pair size increases. Both vLog and HashKV have lower throughput than LevelDB and RocksDB when the KV pair size is 256 B, since the overhead of writing small values to the value store is more significant. Nevertheless, HashKV can benefit from selective KV separation (see Experiment 7). As the KV pair size increases, HashKV also sees increasing throughput. For example, HashKV achieves 15.5× and 2.8× throughput over LevelDB and RocksDB, respectively, for 4-KiB KV pairs. HashKV achieves 2.2-5.1× throughput over vLog for KV pair sizes between 256 B and 4 KiB. The performance gap between vLog and HashKV narrows as the KV pair size increases, since the size of the LSM-tree decreases with fewer KV pairs. Thus, the queries to the LSM-tree of vLog are less expensive. For 64-KiB KV pairs, HashKV has 10.7% less throughput than vLog.

When the KV pair size increases, the total write sizes of LevelDB and RocksDB increase due to the increasing compaction overhead, while those of HashKV and vLog decrease due to fewer KV pairs in the LSM-tree. Overall, HashKV reduces the total write sizes of LevelDB, RocksDB, and vLog by 43.2-78.8%, 33.8-73.5%, and 3.5-70.6%, respectively.

Experiment 5 (Range scans): We compare the range scan performance of KV stores for different KV pair sizes. Specifically, we first load 40 GiB of fixed-size KV pairs, and then issue scan requests whose start keys follow a Zipf distribution with a Zipfian constant of 0.99. Each scan request reads 1 MiB of KV pairs, and the total scan size is 4 GiB. Figure 9 shows the results. HashKV has similar scan performance to vLog across KV pair sizes. However, HashKV has 70.0% and 36.3% lower scan throughput than LevelDB for 256-B and 1-KiB KV pairs, respectively, mainly because HashKV needs to issue reads to both the LSM-tree and the value store and there is also high overhead of retrieving small values from the value store via random reads. Nevertheless, for KV pairs of 4 KiB or larger, HashKV outperforms LevelDB, e.g., by 94.2% for 4-KiB KV pairs. The lower



(a) Throughput (b) Total write size

Figure 10: Experiment 6: Hotness awareness.

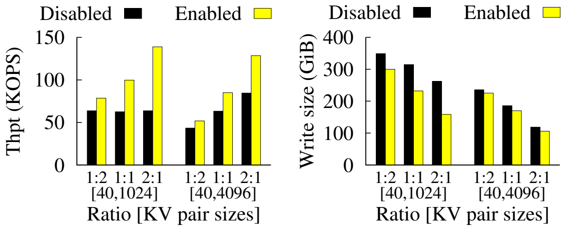
scan performance for small KV pairs is also consistent with that of WiscKey (see Figure 12 in [23]). Note that the read-ahead mechanism (see §3.6) is critical to enabling HashKV to achieve high range scan performance. For example, the range scan throughput of HashKV increases by 81.0% for 256-B KV pairs compared to without read-ahead. We also evaluate the range scan performance of HashKV after we issue update-intensive workloads in [5].

4.4 HashKV Features

We study the two optimizations of HashKV, hotness awareness and selective KV separation, and the crash consistency mechanism of HashKV. We report the throughput in Phase P3 and the total write size. We consider 20% of reserved space to show that the optimized performance of smaller reserved space can match the unoptimized performance of larger reserved space.

Experiment 6 (Hotness awareness): We evaluate the impact of hotness awareness on the update performance of HashKV. We consider two Zipfian constants, 0.9 and 0.99, to capture different skewness in workloads. Figure 10 shows the results when hotness awareness is disabled and enabled. When hotness awareness is enabled, the update throughput increases by 113.1% and 121.3%, while the write size reduces by 42.8% and 42.5%, for Zipfian constants 0.9 and 0.99, respectively.

Experiment 7 (Selective KV separation): We evaluate the impact of selective KV separation on the update performance of HashKV. We consider three ratios of small-to-large KV pairs, including 1:2, 1:1, and 2:1. We set the small KV pair size as 40 B, and the large KV pair size as 1 KiB or 4 KiB. Figure 11 shows the results when selective KV separation is disabled or enabled. When selective KV separation is enabled, the throughput increases by 23.2-118.0% and 19.2-52.1% when the large KV pair size is 1 KiB and 4 KiB, respectively. We observe higher performance gain for workloads with a higher ratio of small KV pairs, due to the high update overhead of small KV pairs stored under KV separation. Also, selective KV separation reduces the total write size by 14.1-39.6% and 4.1-10.7% when the large KV pair size is 1 KiB and 4 KiB, respectively.



(a) Throughput (b) Total write size

Figure 11: Experiment 7: Selective KV separation.

	Disabled	Enabled
Throughput (KOPS)	58.0	54.3
Total write size (GiB)	454.6	473.7

Table 1: Experiment 8: Performance of HashKV with crash consistency disabled and enabled.

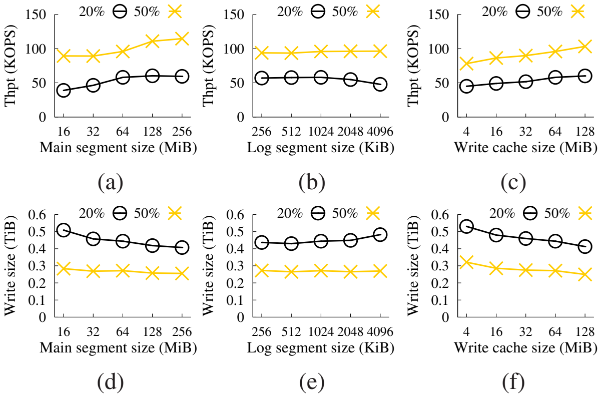
Experiment 8 (Crash consistency): We study the impact of the crash consistency mechanism on the performance of HashKV. Table 1 shows the results. When the crash consistency mechanism is enabled, the update throughput of HashKV in Phase P3 reduces by 6.5% and the total write size increases by 4.2%, which shows that the impact of crash consistency mechanism remains limited. Note that we verify the correctness of the crash consistency mechanism by crashing HashKV via code injection and unexpected terminations during runtime.

4.5 Parameter Choices

We further study the impact of parameters, including the main segment size, the log segment size, and the write cache size on the update performance of HashKV. We vary one parameter in each test, and use the default values for other parameters. We report the update throughput in Phase P3 and the total write size. Here, we focus on 20% and 50% of reserved space.

Experiment 9 (Impact of main segment size, log segment size, and write cache size): We first consider the main segment size. Figures 12(a) and 12(d) show the results versus the main segment size. When the main segment size increases, the throughput of HashKV increases, while the total write size decreases. The reason is that there are fewer segment groups for larger main segments, so each segment group receives more updates in general. Each GC operation can now reclaim more space from more updates, so the performance improves. We see that the update performance of HashKV is more sensitive to the main segment size under limited reserved space. For example, the throughput increases by 52.5% under 20% of reserved space, but 28.3% under 50% of reserved space, when the main segment size increases from 16 MiB to 256 MiB.

We next consider the log segment size. Figures 12(b)



(a) (b) (c)

(d) (e) (f)

Figure 12: Experiment 9: Throughput and total write size of HashKV versus the main segment size ((a) and (d)), the log segment size ((b) and (e)), and the write cache size ((c) and (f)).

and 12(e) show the results versus the log segment size. We see that when the log segment size increases from 256 KiB to 4 MiB, the throughput of HashKV drops by 16.1%, while the write size increases by 10.4% under 20% of reserved space. The reason is that the utilization of log segments decreases as the log segment size increases. Thus, each GC operation reclaims less free space, and the performance drops. However, when the reserved space size increases to 50%, we do not see significant performance differences, and both the throughput and the write size remain almost unchanged across different log segment sizes.

We finally consider the write cache size. Figures 12(c) and 12(f) show the results versus the write cache size. As expected, the throughput of HashKV increases and the total write size drops as the write cache size increases, since a larger write cache can absorb more updates. For example, under 20% of reserved space, the throughput of HashKV increases by 29.1% and the total write size reduces by 16.3% when the write cache size increases from 4 MiB to 64 MiB.

5 Related Work

General KV stores: Many KV store designs are proposed for different types of storage backends, such as DRAM [1, 13, 14, 21], commodity flash-based SSDs [8,9,20,23], open-channel SSDs [34], and emerging non-volatile memories [25, 40]. The above KV stores and HashKV are designed for a single server. They can serve as building blocks of a distributed KV store (e.g., [28]).

LSM-tree-based KV stores: Many studies modify the LSM-tree design for improved compaction performance. bLSM [33] proposes a new merge scheduler to prevent compaction from blocking writes, and uses Bloom filters for efficient indexing. VT-Tree [35] stitches already sorted blocks of SSTables to allow lightweight compaction overhead, at the expense of incurring fragmentation. LSM-trie [39] maintains a trie structure and organizes KV pairs by hash-based buckets within each SSTable. It also organizes large Bloom filters in clustered disk blocks for efficient I/O access. LWC-store [41] decouples data and metadata management in compaction by merging and sorting only the metadata in SSTables. SkipStore [42] pushes KV pairs across non-adjacent levels to reduce the number of levels traversed during compaction. PebblesDB [30] relaxes the restriction of keeping disjoint key ranges in each level, and pushes partial SSTables across levels to limit compaction overhead.

KV separation: WiscKey [23] employs KV separation to remove value compaction in the LSM-tree (see §2.2). Atlas [18] also applies KV separation in cloud storage, in which keys and metadata are stored in an LSM-tree that is replicated, while values are separately stored and erasure-coded for low-redundancy fault tolerance. Cocytus [43] is an in-memory KV store that separates keys and values for replication and erasure coding, respectively. HashKV also builds on KV separation, and takes one step further to address efficient value management.

Hash-based data organization: Distributed storage systems (e.g., [10, 24, 38]) use hash-based data placement to avoid centralized metadata lookups. NVMKV [25] also uses hashing to map KV pairs in physical address space. However, it assumes sparse address space to limit the overhead of resolving hash collisions, and incurs internal fragmentation for small-sized KV pairs. In contrast, HashKV does not cause internal fragmentation as it packs KV pairs in each main/log segment in a log-structured manner. It also supports dynamic reserved space allocation when the main segments become full.

6 Conclusion

This paper presents HashKV, which enables efficient updates in KV stores under update-intensive workloads. Its novelty lies in leveraging hash-based data grouping for deterministic data organization so as to mitigate GC overhead. We further enhance HashKV with several extensions including dynamic reserved space allocation, hotness awareness, and selective KV separation. Testbed experiments show that HashKV achieves high update throughput and reduces the total write size.

Acknowledgements

The work was supported by the Research Grants Council of Hong Kong (CRF C7036-15G) and National Nature Science Foundation of China (61772484 and 61772486). Yongkun Li is the corresponding author.

References

[1] Redis. http://redis.io.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. of USENIX ATC*, 2008.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proc. of ACM SIGMETRICS*, 2012.

[4] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proc. of USENIX OSDI*, 2010.

[5] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. Technical report, CUHK, 2018. http://www.cse.cuhk.edu.hk/~pclee/www/pubs/tech_hashkv.pdf.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of USENIX OSDI*, 2006.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of ACM SoCC*, 2010.

[8] B. Debnath, S. Sengupta, and J. Li. Flash-Store: High Throughput Persistent Key-value Store. *Proc. of VLDB Endowment*, 3(1-2):1414–1425, Sep 2010.

[9] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage. In *Proc. of ACM SIGMOD*, 2011.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of ACM SOSP*, 2007.

[11] R. Escriva. HyperLevelDB. https://github.com/rescrv/HyperLevelDB/.

[12] Facebook. RocksDB. https://rocksdb.org.

[13] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of USENIX NSDI*, 2013.

[14] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124), Aug 2004.

[15] S. Ghemawat and J. Dean. LevelDB. https://leveldb.org.

[16] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang. Efficient Identification of Hot Data for Flash Memory Storage Systems. *ACM Trans. on Storage*, 2(1):22–40, Feb 2006.

[17] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *Proc. of IEEE IISWC*, 2008.

[18] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's Key-value Storage System for Cloud Data. In *Proc. of IEEE MSST*, 2015.

[19] J. Lee and J.-S. Kim. An Empirical Study of Hot/Cold Data Separation Policies in Solid State Drives (SSDs). In *Proc. of ACM SYSTOR*, 2013.

[20] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. of ACM SOSP*, 2011.

[21] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proc. of USENIX NSDI*, 2014.

[22] Linux Raid Wiki. RAID setup. https://raid.wiki.kernel.org/index.php/RAID_setup.

[23] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. In *Proc. of USENIX FAST*, 2016.

[24] J. MacCormick, N. Murphy, V. Ramasubramanian, U. Wieder, J. Yang, and L. Zhou. Kinesis: A New Approach to Replica Placement in Distributed Storage Systems. *ACM Trans. on Storage*, 4(4):11, 2009.

[25] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *Prof. of USENIX ATC*, 2015.

[26] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proc. of ACM SOSP*, 1997.

[27] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proc. of USENIX FAST*, 2012.

[28] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and enkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proc. of USENIX NSDI*, 2013.

[29] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[30] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proc. of ACM SOSP*, 2017.

[31] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Trans. on Computer Systems*, 10(1):26–52, Feb 1992.

[32] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proc. of USENIX FAST*, 2014.

[33] R. Sears and R. Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proc. of ACM SIGMOD*, 2012.

[34] Z. Shen, F. Chen, Y. Jia, and Z. Shao. DIDACache: A Deep Integration of Device and Application for Flash Based Key-Value Caching. In *Proc. of USENIX FAST*, 2017.

[35] P. J. Shetty, R. P. Spillane, R. R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building Workload-Independent Storage with VT-Trees. In *Proc. of USENIX FAST*, 2013.

[36] threadpool. http://threadpool.sourceforge.net/.

[37] TPC. TPC-C is an On-Line Transaction Processing Benchmark. http://www.tpc.org/tpcc/.

[38] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proc. of USENIX OSDI*, 2006.

[39] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *Proc. of USENIX ATC*, 2015.

[40] F. Xia, D. Jiang, J. Xiong, and N. Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proc. of USENIX ATC*, 2017.

[41] T. Yao, J. Wan, P. Huang, X. He, Q. Gui, F. Wu, and C. Xie. A Light-weight Compaction Tree to Reduce I/O Amplification toward Efficient Key-Value Stores. In *Proc. of IEEE MSST*, 2017.

[42] Y. Yue, B. He, Y. Li, and W. Wang. Building an Efficient Put-Intensive Key-Value Store with Skip-Tree. *IEEE Trans. on Parallel and Distributed Systems*, 28(4):961–973, Apr 2017.

[43] H. Zhang, M. Dong, and H. Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *Proc. of USENIX FAST*, 2016.